

THE UNIVERSITY OF MICHIGAN
COLLEGE OF LITERATURE, SCIENCE, AND THE ARTS
Computer and Communication Sciences Department

Technical Report

A SELF-DESCRIBING AXIOMATIC SYSTEM AS A SUGGESTED
BASIS FOR A CLASS OF ADAPTIVE THEOREM PROVING MACHINES

Thomas H. Westerdale

supported by:

Department of Health, Education, and Welfare
National Institutes of Health
Grant No. GM-12236-03
Bethesda, Maryland

and

Office of Naval Research
Contract No. N00014-67-A-0181-0011
Washington, D. C.

and

U. S. Army Research Office (Durham)
Grant No. DA-31-124-ARO-D-483
Durham, North Carolina

administered through:

OFFICE OF RESEARCH ADMINISTRATION

ANN ARBOR

March 1969

Distribution of This Document is Unlimited.

ERRATA SHEET

Page 129, Line 12 should read:

We shall now show that for any theorem of Form $\Theta \supset \alpha$, the formula

Page 129, Line 13 should read:

$\sim T(\alpha)$ is not provable. Suppose $\Theta \supset \alpha$ is a theorem. Then we can

Page 129, Line 22 should read:

by rule 18 from theorem $\Theta \supset \alpha$.

Page 129, Line 23 should be deleted:

Page 130, After Line 4 Insert:

Let us call an expression anomalous if it contains a subexpression of form (β, γ, δ) in which occurs a variable ξ such that $\xi \triangleleft \delta$ holds but ξ is not free in (β, γ, δ) . Such anomalous expressions can never occur inside theorems. Now if α is a well formed formula which is not an anomalous expression, then $\Theta \supset \alpha$ is provable. Hence we have shown that for any well formed formula α which is not anomalous, $\sim T(\alpha)$ is not provable. It would have been much more natural (and quite easy) to have originally defined the class of well formed expressions (and of well formed formulae) so as to exclude anomalous expressions. (We could even have defined the class of expressions so as to exclude anomalous expressions.) With such a natural definition, an expression α is well formed if and only if it occurs as a subexpression of some theorem; that is, if and only if

$$\begin{aligned} \mathfrak{H}(z, x) & (\text{expression}(z) \wedge \text{variable}(x) \wedge x \triangleleft z \wedge \text{type}(x) = \\ & \text{expertype}(\alpha) \wedge T(\text{Sf}(x, \alpha), z)) \end{aligned}$$

holds. This is what is usually meant by a well formed expression.

Page 130, After the Last Line Insert:

If *anom* is a predicate expression such that *anom*(σ) holds if and


Errata Sheet (Cont.)


and only if σ names an anomalous expression, then $T(x) \supset \sim anom(x)$ is provable by a tedious proof which, in outline, is something like the proof of $T(x) \supset F(x)$. Thus if σ is an anomalous expression, $\sim T(\sigma)$ is provable by the same argument as above.


Page 214, Delete Lines 16, 17, 18, and 19.

PREFACE

Throughout the original manuscript colored symbols were used to represent quoted symbols. Use of colored symbols permits a more intuitive and economical abbreviation scheme than does the use of quotation marks. The abbreviations discussed in Section 2.1.2.6 would be confusing if quotation marks were used in place of color. In this particular copy it has been impossible (for typographical reasons) to use colored symbols. We have therefore employed the following convention:

Symbols blue in the original manuscript are here surrounded by a balloon  .

Symbols red in the original manuscript are here surrounded by a double balloon  .

Symbols green in the original manuscript are here surrounded by a triple balloon  .

The text has not been changed in this copy. Therefore symbols inside the double balloons are referred to as red symbols, etc. The reader may wish to clarify some of the more complicated formulae (particularly those in Section 2.2.2.) by writing them out in the original color notation.

The description of our axiomatic system begins in Section 2.1.2. Section 2.1.1. describes the relationship of our axiomatic system to formal arithmetic. Section 2.1.1. does give an intuitive overview of the system, but some of the arguments there are more complete than is required for such an overview. The reader may wish on the first reading to skip the more tedious portions of Section 2.1.1., particularly the more tedious portions of 2.1.1.9. Evidence for claims made in Section 2.1.1. is frequently given in footnotes. Use has been made, in these notes, of notation explained in Section 2.1.2, so the reader should not be surprised to find

certain portions of these notes unintelligible until after he has read Section 2.1.2.

The conclusion of Section 2.2.1 is that our axiomatic system is consistent. The reader is assured that this fact, the consistency of our system, is the only thing from Section 2.2.1 that is used in the other sections. The reader who does not wish to read the somewhat tedious Section 2.2.1, and who can believe that the system is consistent, may proceed to the short Section 2.2.2 (where other logical properties of the system are concisely developed) without fear that he has missed something that will be referred to later.

It should also be noted that the reader who has read no more than Section 1 will be able to understand (though perhaps not believe) much of the conclusion of section 3.8 .

The present work is an outgrowth of certain investigations in the theory of Adaptive Systems as developed by Professor John H. Holland. The approach used here is the approach which Professor Holland has developed for more general cases. My original inspiration was Professor Holland's Iterative Circuit Computer which resembles the memory nets described here (taking his generators to be my formula nodes). Professor Holland provided the key to the scheme described in this paper when he suggested that heuristics might be regarded as "rules of inference with some conditions missing." The purpose of this work is to provide a scheme which allows the theory to be applied in a theorem proving environment.

I would like to thank each member of my doctoral committee for his guidance and aid.

Professor Peter G. Hinman guided me through logical arguments required

in Section 2.2, the section which presents the major results of this work. It was necessary for Professor Hinman to give me a course in Logic, suggest various approaches to my problem, and demolish many of my arguments before the arguments presented in Section 2.2 could be completed.

Professor Arthur W. Burks showed me the necessity of making the arguments given in Section 2.2. For example, he pointed out that if statement (B) Section 2.2.2 holds, my system is inconsistent. Since I thought at that time that Statement (B) held, I realized my argument was full of holes.

Professor Bruce W. Arden and Professor Bernard A. Galler aided me in the implementation aspects of this project. They pointed out areas of difficulty which, with my limited programming experience, I might otherwise have missed. I was, for example, entirely oblivious of the difficulties discussed in Section 3.3.5 until I was forced to think in detail about storage methods.

I would like to express my gratitude for the privilege of working with the members of the Logic of Computers Group at The University of Michigan. I would like to thank the members of the Group and of the Computer and Communication Sciences Department for providing a stimulating environment, for asking many helpful questions, and for making many helpful suggestions.

I am especially indebted to the late Professor Gordon Peterson for shielding me from the sort of strict calendaring of courses and exams which stifles a person's education. His advice in matters both academic and bureaucratic was extremely helpful.

I would like to thank my instructors and fellow students of The University of Michigan Department of Botany for giving me the understanding of biological processes that is important for Adaptive Systems study.

I would like especially to thank Professor K. L. Jones of The University of Michigan Botany Department who was a constant source of inspiration during the nine years he guided my education as my instructor and counselor.

This work was supported by the National Institutes of Health, the Army Research office, and the Office of Naval Research.

TABLE OF CONTENTS

PREFACE	ii
LIST OF FIGURES	ix
1. INTRODUCTION	1
1.1 Purpose of the Paper	1
1.2 The Difficulty in Making "Small" Enough Changes	2
1.3 Our Approach: The Meta and Object Level	3
1.4 The Machine's Environment	6
1.5 Organization of the Machine's Memory, an Example	7
1.6 The Effector Acting Upon the Memory Net	13
1.6.1 Principles of Effector Action	13
1.6.2 Use of the Heuristics that are in the Net: an Example	14
1.6.3 Refining a Proof	17
1.6.3.1 The General Scheme-- <i>refineproof</i>	17
1.6.3.2 <i>prove</i> --and the Problem of Loose Ends	20
1.6.4 Saving Examples of Derivations Employing Heuristics	22
1.7 Generation of Heuristics	26
1.8 Questions Beyond the Scope of the Paper	28
2. THE FORMAL SYSTEM	30
2.1 Basic Structure of the System	30
2.1.1 Overview of System	30
2.1.1.1 Plan of Discussion	30
2.1.1.2 Pertinent Properties of Formal Arithmetic	32
2.1.1.3 Addition of ι (representability of individual functions)	36
2.1.1.4 Addition of λ (ordinary-names of functions and relations)	38
2.1.1.5 Addition of label, cond, and pcond (algorithmic-names of functions and relations)	38
2.1.1.6 Elimination of + and \cdot	43
2.1.1.7 An Operation on S-expressions: Addition of * and nil	44
2.1.1.8 Addition of Pv, Iv, Pfvb, Ifvb, newpv, newiv, newpfvb, and newifvb; subtraction of tv, I, Pf, and If	46
2.1.1.9 Elimination of S and \emptyset and addition of qu	49
2.1.1.10 Our Axiom Set	55
2.1.2 Expressions and Their Abbreviation	57
2.1.2.1 Motivation for Abbreviation	57
2.1.2.2 Definitions of Classes of Expressions of our Language	58
2.1.2.3 Format for Abbreviation Rules	63
2.1.2.4 Abbreviations of a Single Color with no "Defined" Symbols (the first 12 rules)	63
2.1.2.5 "Defined" Symbols (Rule 13)	67
2.1.2.6 Abbreviations using Colored Symbols (Rules 14-17)	69
2.1.2.7 Comma vs. Dot Notation	73
2.1.2.8 Reading the Expressions in the Tables and Text	73

TABLE OF CONTENTS (Cont.)

2.1.3	Well Formedness	77
2.1.4	The Axiomatic System	80
2.1.4.1	Certain Functions and Relations	80
2.1.4.2	The Meta Level	86
2.1.4.3	The Axioms (Table 4)	91
2.1.4.4	The Rules of Inference (Table 5)	93
2.1.4.5	Proofs and Theorems	96
2.1.4.6	Summary	97
2.2	Formal Arguments	98
2.2.1	Consistency	98
2.2.1.1	Generation of Function Expressions	98
2.2.1.2	Preservation of Consistency while Making Identifications between Object and Meta Levels. (Completion of Consistency Argument)	120
2.2.2	Incompleteness	126
3.	IMPLEMENTATION	131
3.1	Purpose of Section 3	131
3.2	Characterization of the Subclass of Machines	132
3.3	Structure of Memory	133
3.3.1	Basic Plan of the Memory Net	133
3.3.2	Some LISP Functions on Bug Values	136
3.3.3	Condition on Derivation Nodes (Rules and Heuristics)	139
3.3.4	Net Changing Functions	142
3.3.5	Storage of Memory Nets-Difficulties	145
3.3.6	Patching	148
3.3.7	Tagging and Garbage Collection	150
3.4	Other Functions which Change Net Structure	151
3.4.1	Kinds of Functions to be Discussed	151
3.4.2	Searching	152
3.4.3	Functions on Two Nets	154
3.5	LISP Structure of the Effector	158
3.6	Refining a Proof	158
3.6.1	The Task of <i>refineproof</i>	158
3.6.2	The Use of T-tags and H-tags	160
3.6.3	The Task of <i>prove</i>	161
3.6.4	Example: A Heuristic which is a Composition of Two Rules	162
3.6.5	Less Trivial Situations	167
3.6.6	<i>parameteretreegenerate</i>	168
3.6.7	Suppose the Model Fails at Some Point	171
3.7	General Considerations	173
3.8	Conclusion: Discussion of Adaptation	174
3.9	Postscript: Other Object Theories	179
4.	TABLES	187
4.1	Table 1. Alphabet	187
4.2	Table 2. Basic Recursive Functions	188
4.3	Table 3. Defined Complete, Recursive Functions of a General Nature	190
4.4	Table 4. Axioms	196

TABLE OF CONTENTS (Cont.)

4.5	Table 5. Rules of Inference	199
4.6	Table 6. Defined Complete, Recursive Functions of a Specific Nature	203
4.7	Table 7. Definition of T and Immediate Consequences	210
4.8	Table 8. Non-Recursive Definitions Especially Useful for Meta-Theorems. Some Immediate Consequences	211
4.9	Table 9. Definitions for Handling Recursive Functions; <i>apl</i>	215
4.10	Table 10. Examples	218
4.10.1	Example 1. Reflexivity and Transitivity of =	218
4.10.2	Example 2. Proof of $\sim atom(x) \equiv a(x)*d(x)$ (basic theorem for <i>a</i> and <i>d</i>)	218
4.10.3	Example 3. Some More Theorems about <i>a</i> and <i>d</i> ; an Alternative Induction Axiom	221
4.10.4	Example 4. Course of Values Induction (and a Corollary)	222
4.10.5	Example 5. Generation of some Labeled Functions, e.g., <i>maplist</i>	224
4.10.6	Example 6. The Predecessor Function	234
4.10.7	Example 7. The μ Schema	236
4.10.8	Example 8. $T(x) \supset F(x)$	237
4.10.9	Example 9. Sketch of Proof of $T(x) \supset T(apl(x))$	238
4.11	Table 11. Implementation Routines--Effector Algorithms Used in Section 3	242
4.11.1	Basic Functions and Notation	242
4.11.2	Routines which Return a Bug Value	244
4.11.3	Routines which Return a Bug Value Paired with a Sequence of Pairs	247
	REFERENCES	252

LIST OF FIGURES

Figure 1	9
Figure 2	10
Figure 3	11
Figure 4	12
Figure 5	16
Figure 6	23
Figure 7	24
Figure 8	25
Figure 9	143
Figure 10	144
Figure 11	148
Figure 12	160
Figure 13	164
Figure 14	165
Figure 15	170
Figure NSS1	182
Figure NSS2	182
Figure NSS3	183
Figure NSS4	183
Figure NSS5	184
Figure NSS6	184

1. INTRODUCTION

1.1 Purpose of the Paper

The overall goal of this paper is to argue for the thesis that we can arrive at a more general approach to adaptive theorem proving if we first formulate an axiomatic theory which relates structure of computer programs to their performance. A major part of the argument will consist in exhibiting an axiomatic theory which possesses the required features.

Until now, adaptation in theorem proving machines has been rather limited. For example, Newell, Shaw, and Simon's General Problem Solver has a fixed set of heuristics (i.e., tests to see which strategy is to be followed) which are employed successively to construct a proof. The machine can adapt by changing the probabilities with which the various heuristics are employed, but the set of heuristics it may use remains the same set that the programmer put into the machine. It has no capacity (or, in later models, very limited capacity) to generate new heuristics.

In this paper we shall show how an axiomatic theory relating structure of computer programs to their performance can be used as the core of an adaptive theorem prover which has a general capability of generating new heuristics. First we shall design a language suitable for talking about computer (LISP) programs. We shall then construct a set of axioms and rules of inference such that the theorems of this axiomatic system have natural interpretations as statements about LISP programs.

In these theorems, LISP programs appear as long expressions which behave syntactically as names of partial recursive functions.

Let us write ϕ , θ , and ψ as abbreviations for whole LISP programs. Then $\phi(\alpha)$ stands for the output of the program ϕ when

given the input α , just as in normal functional notation. For example, if $(\theta(x) = 0) \supset (\phi(x) = \psi(x))$ is a theorem, then the statement " ϕ and ψ give identical outputs for identical inputs as long as θ gives output zero for those inputs" will be a true statement.

These statements include statements relating the structure of LISP programs to their performance. Finally we shall discuss one method by which this axiomatic system may be implemented as the core of an adaptive theorem proving machine.

The main section of this paper will be devoted to a formal description of the axiomatic system. We shall not go into this formalism in the introduction. We simply mention that the system is as powerful as formal arithmetic. In the main section we shall show how the axiomatic system may be derived from formal arithmetic by a series of consistency-preserving transformations. In addition to showing consistency of the system, we shall give a Gödel-type proof of incompleteness. Following the main section will be a section explaining how the system might be implemented as the core of an adaptive theorem prover. The remainder of this introduction will outline some of the significant characteristics of the sort of adaptive theorem prover we have in mind.

1.2 The Difficulty in Making "Small" Enough Changes

For a machine to adapt it must change its structure bit by bit, checking after each small change to see whether the change has been an improvement or a pejection. A radical change is almost always pointless.

But what does it mean to make a small change in structure, as opposed to a radical change? In Friedberg's program generating machine [Friedberg, 1958], the "structure" was a computer program written in an assembly language.

The structure was modified by changing various instructions. The performance before the change was then compared with the performance after the change. A "small" change for Friedberg meant a change of only one or two instructions (as opposed to a change of many instructions). Such a definition of "small" is meaningless since his "small" structural changes tended to produce radical performance changes. What is needed is a theory of structural change which predicts which types of changes will produce "small" performance changes and which ones "large" performance changes. As yet no adaptive machines have employed any such theory.

Of course the relationship between structural changes in programs and concomitant performance changes is much too complicated to be represented as a relationship between "magnitude" of the change.

McCarthy has attacked this problem in the following manner. He has first developed a programming language (LISP 1.5) which lends itself to analysis of this sort. A LISP program is a symbolic representation of a partial recursive function. The "data" or "inputs" to the program are names for the function's arguments. The name of the value of the function for those arguments is the "output". McCarthy has then developed a theory which tells how to set up certain algorithms which, when applied to the function representations, tell us, among other things, over what sets two functions are identical. However, he gives no usable mechanical procedure for generating the various algorithms. (The set of algorithms we want is not a recursive set. In our system the set is defined by a generation procedure discussed in Section 2.2.1.1)

1.3 Our Approach: The Meta and Object Level

Our approach will be more general. We design an axiomatic system, S, the theorems of which have natural interpretations as statements about LISP programs. These theorems, once proved, can be used in modifying LISP programs in any

desired way. (Actually we use a modified LISP, modified in several essential ways.) A machine can use these theorems as a basis on which to make adaptive changes. We shall argue that there exist useful theorem proving machines which make adaptive changes on the basis of these theorems. We shall do this by describing such a machine in sufficient detail to demonstrate its characteristic properties.

This machine will make two kinds of adaptive changes. First, it will generate new appropriate heuristics. Second, it will change the probabilities with which the various heuristics are employed. It is chiefly in the ability to generate new heuristics that this machine differs from such machines as the Newell, Shaw, and Simon General Problem Solver. We shall, in this section (Sec. 1.3), describe in a general way the method by which new heuristics are generated. Later in the introduction we shall discuss in some detail how our machine changes the probabilities with which the various heuristics are employed.

Our machine will be making the adaptive changes in a theorem proving environment. An essential part of the machine will be a set of axioms and rules of inference for the system in which we wish the theorems proved. (The machine will prove theorems in a manner similar to that of Newell, Shaw, and Simon's General Problem Solver.) Let us call this the object system, and let us call the language in which the theorems are written the object language. The axioms are written in the object language. The rules of inference are like little LISP programs, each representing a function which maps theorems to theorems. The rules of inference are written in a LISP-like meta language.

In addition, our machine will need other expressions written in the meta language. It will need expressions which look like rules of inference, but which are only sometimes valid. These we call heuristics. They will be associated with rules of inference in such a way that the successful

application of a heuristic will imply, for example, that one of a particular set of rules of inference is likely to be successful. A heuristic is a quick check to see whether a strategy is a good one. (The machine will, of course, assign weights to various heuristics according to past success.) A well-adapted machine will have a well-organized hierarchy of heuristics which classify rules of inference into useful overlapping classes. We will see more complicated uses of heuristics later. (We will also see how Newell, Shaw, and Simon's heuristics can be written in our notation.) Some of the simpler heuristics may be thought of as merely rules of inference with some of the required conditions missing. The situation is not always so simple, but there will always be some useful relationship between the performance of the heuristic and the performance of the rules of inference to which it is related. Now if our machine only had a good set of statements relating structure of LISP programs to performance, it could generate new heuristics directly from the rules of inference by making indicated structural changes. (It could also generate new rules of inference from old rules of inference, and new heuristics from old heuristics.)

Suppose now that the object system is the system S whose theorems have natural interpretations as statements about LISP programs. (Some of these statements relate structure of LISP programs to their performance.) Then as our theorem proving machine worked, it would produce these statements which would tell how to modify our rules of inference and heuristics to attempt to improve the performance of our machine. In fact, we can even write rules of inference in a format such that they themselves become theorems of the system S. They will be theorems whose natural interpretations are statements of a form something like, "If such and such is a theorem, then so and so is a theorem." In fact we shall see later that

heuristics may be written in a similar format so that they become formulae (but not theorems) of S which have a form similar to the form of rules of inference. Thus the theorems of the object system (now the system, S), the rules of inference, and the heuristics are all written in the same language, the object language of the system, S. The language of our meta and object levels is identical. We will see how this identification of meta and object level is exactly the same as the well-known identification made in formal arithmetic by means of Gödel numbering. In our system, however, we have theorems which are statements about practical computer programs.

1.4 The Machine's Environment

Likening the machine to, say, a mathematics professor's graduate assistant, one sees that the environment of the machine is not merely the mathematical system in which the theorems are to be proved, but includes also the value placed on theorems and completed proofs by the professor (or, in the machine's case, the user) and the ordering of the problem sequence presented. This is important, for it means that the machine, in adapting to take advantage of regularities in its environment, can take advantage of regularities both in the mathematical system itself and in the mind of the user. Any machine not taking advantage of this second class of regularities is ignoring vital information and may stand little chance of learning. (This point was missed by Amarel [Amarel 1962], whose machine may have been taking advantage of regularity in the user's mind rather than, as Amarel claims, regularity in the mathematical system.) The graduate assistant takes advantage of such regularity in the professor's mind when he produces "elegant" proofs rather than tedious ones, and when he becomes proficient in techniques for proving important theorems, ignoring

whole classes of trivial unapplicable theorems. ("Elegance" and "importance" are concepts in the professor's mind, not in the mathematical system.) The student takes advantage of such regularity when he looks first at most recent lessons in his search for theorems to use in homework problems. (It will not be absolutely necessary that the user use programmed learning techniques in developing machine competence in the environment in question, but it may frequently be advisable. Similarly for the professor developing student competence.) One consequence of the above point of view is that the sequence of problems and the rewards given are ultimately determined by the user.

Thus in a vague sense the environment will be a sequence of "to prove" problems presented by the user together with the rewards given by the user for the completed proofs produced by the machine. The environment will be more regular and the machine's task simpler accordingly as the user grades the problems in difficulty and makes them interrelated.

1.5 Organization of the Machine's Memory, an Example

We seek a machine "intelligent" enough to operate in this environment. Our approach is to construct a machine which attacks problems somewhat the way humans attack them.

This approach presupposes some knowledge of human thinking. Our knowledge is obviously meager, but I have found Polya's How to Solve It [Polya 1945] an invaluable aid. His suggestions are illuminating:

"Do you know a related problem? ... Here is a problem related to yours and solved before. ... Could you use its result?
Could you use its method?"

The human is here taking advantage of the very regularities in the environment we have been mentioning. Our machine must then be prepared to

remember both results and methods of problems solved (together with their values, determined by rewards given by the user) and to find similarities between them.

One way to do this is to remember all previous proofs (proofs which accumulate little reward are, of course, actually thrown away later) and the rules of inference used at each step. Our machine must also remember which heuristics were used to arrive at the proof, but let us ignore this for the moment. Since each line of the proof is a theorem, the machine need only remember, for each of these theorems, the rule used to derive the theorem, the theorems to which the rule was applied, and certain parameter values the purpose of which we shall make clear below. In other words, for each theorem, the machine remembers its immediate derivation.

We now give an example of such an immediate derivation and indicate how it is remembered inside the machine. In this example, for simplicity, the theorems and parameter values will be written in the language of propositional calculus, and the rules will be written in English. (In our machine both theorems and rules are written in the object language of the system, S, but we shall not discuss that formalism here.)

THEOREMS

Thm. 1: $(p \supset (q \supset p)) \supset ((p \supset q) \supset (p \supset p))$

Thm. 2: $p \supset (q \supset p)$

Thm. 3: $(p \supset q) \supset (p \supset p)$

Thm. 4: $(p \supset (q \supset p)) \supset (p \supset p)$

Thm. 5: $p \supset p$

RULES

Rule 1: (Modus ponens) If α and β are well-formed formulae of the

propositional calculus, and if both $\alpha \supset \beta$ and α are theorems, then β is a theorem.

Rule 2: (Substitution) If α is a theorem of the propositional calculus, β is a propositional variable, and γ is a well-formed formula of the propositional calculus, then the formula resulting from uniform substitution of γ for β in α is a theorem.

PARAMETER VALUES

Value 1: q

Value 2: $q \supset p$

Now suppose Theorem 4 was derived directly from Theorem 3 by Rule 2. For Rule 2 to have been properly applied in this case, the parameters α , β , and γ in the statement of the rule must have had values $(p \supset q) \supset (p \supset p)$, q , and $q \supset p$ respectively. The machine stores this information in the following net-like structure.

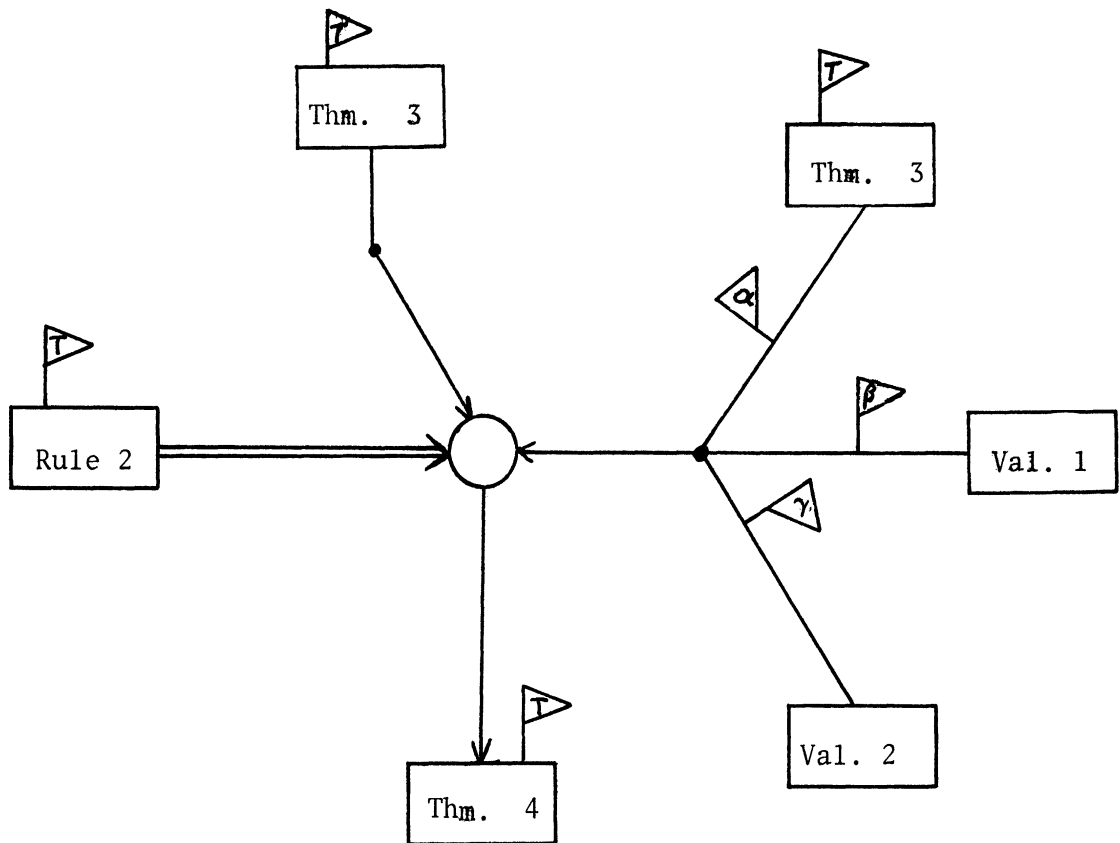


Figure 1.

The circle is called a derivation node. The rectangles are called formula nodes. The dots are called antecedent nodes (the left one) or parameter value nodes (the right one). The triangles are called flags. Flags with a T on them (called tag-type flags) fly from all formula nodes whose contents are already-proved theorems or legitimate rules of inference. The antecedent node is connected to the theorem or theorems to which the rule was applied. The parameter value node is connected to the values (three of them in this case) for the parameters in the statement of the rule. The lines connecting the parameter node to the values have flags (called parameter-type flags) which indicate which value goes with which parameter. The presence of these flags indicates that the parameter value node is indeed a parameter value node and not an antecedent node.

Similarly, suppose Theorem 3 was derived directly from Theorems 1 and 2 by Rule 1; then the machine stores

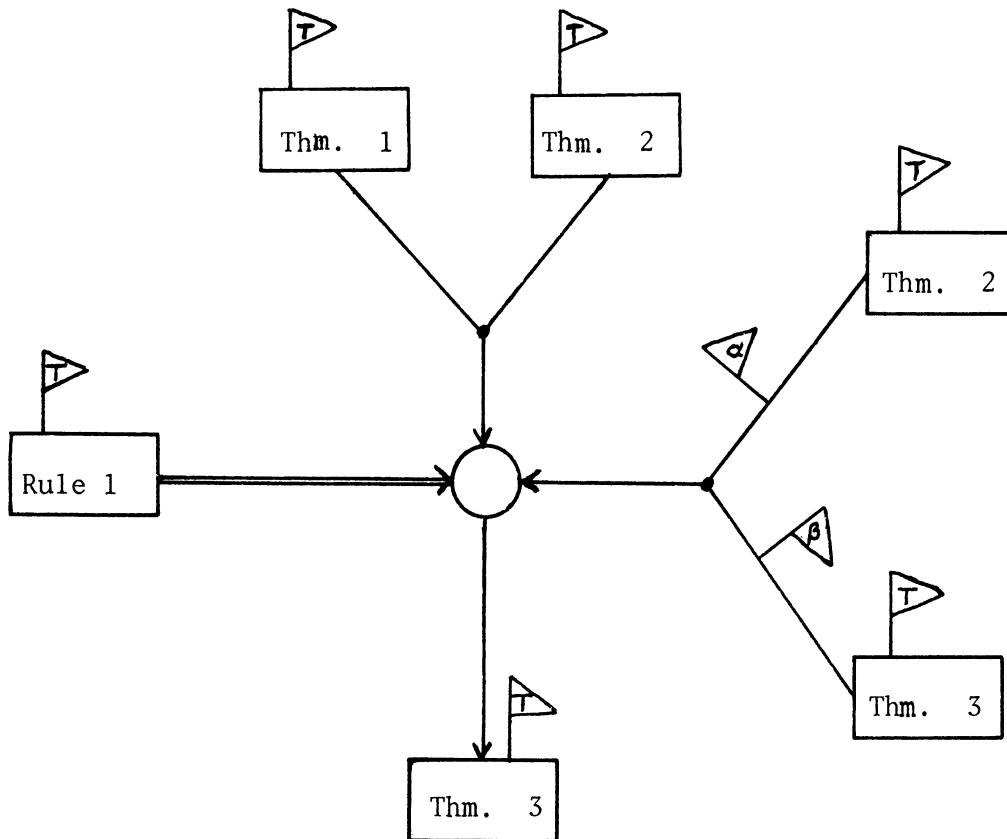


Figure 2.

Note: The values for parameters α and β in Rule 1 are respectively the formula which is Theorem 2 and the formula which is Theorem 3.

Similarly, suppose Theorem 5 was derived directly from Theorems 4 and 2 by Rule 1; then the machine stores

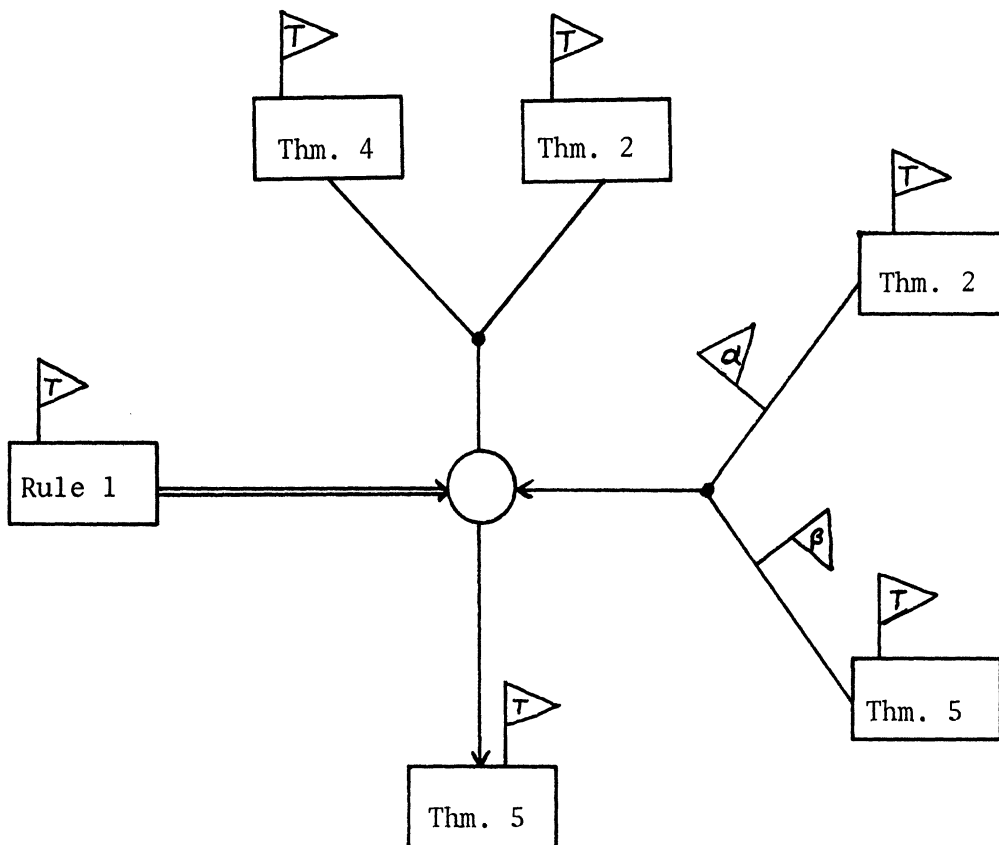
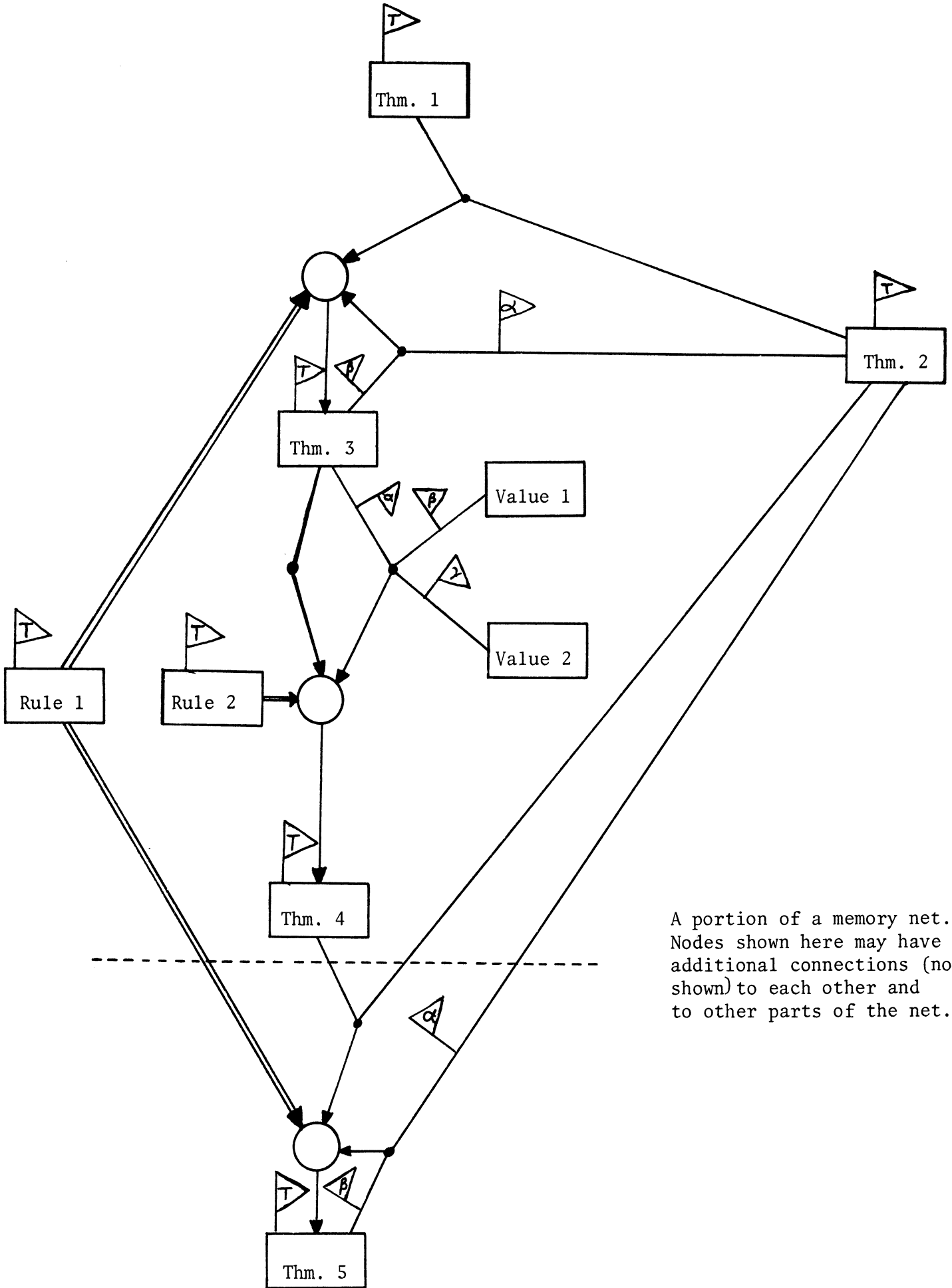


Figure 3.

Of course if the machine wants to store all three derivations it can economize on space by not repeating identical nodes as we have above. By such economy the resulting net becomes as in Figure 4.

The machine stores all such derivations together in one interconnected net called the memory net. This net will contain all the axioms, rules, previously proved theorems, etc., which the machine wants to remember.



A portion of a memory net. Nodes shown here may have additional connections (not shown) to each other and to other parts of the net.

Figure 4.

To construct a new derivation the machine must select the proper rule, theorems, and parameter values from the nodes already in the memory net, then construct arrows and nodes connecting these in order that the desired derivation is included in the net. In one sense, then, the memory net may be regarded as a passive part of the machine which is acted upon by the active part of the machine called the effector. The effector performs the searches for the proper rules, theorems, and parameter values, and performs the construction operations which alter the structure of the net.

1.6 The Effector Acting Upon the Memory Net

1.6.1 Principles of Effector Action. A major problem, for the effector, is finding the proper nodes to connect in order to derive the desired theorems. (Eg. in Fig. 4, to derive Thm. 4, the effector must find nodes containing Rule 2, Thm. 3, Value 1, and Value 2.) We will describe a class of possible search and connection algorithms in some more detail in the implementation section. For now we will merely mention some principles which guide the search technique.

The first principle is that the nodes the effector needs to connect will probably already be fairly close together in the net. Thus if the effector has found a node which it suspects is one of the nodes it wants, it looks nearby to find the other nodes and then checks to see if the proper connections can be made. What do we mean by nearby? In addition to the tag-type flags and parameter-type flags mentioned earlier, there are various sorts of value-type flags (not shown in the preceding figures) attached to nodes and lines between nodes. These flags contain numbers which are used by the effector in searching and constructing. One such flag on a line connecting two nodes gives a measure of the "distance" between the two.

"Distance" between two nodes not directly connected can be determined from the distance between successive nodes along a path connecting the two. The effector conducts several different sorts of searches. For each sort of search there is a different kind of value-type flag and hence a different "distance" measure over the net.

The second principle is that the effector tends to look first at nodes which have been useful in making constructions in the past. Value-type flags attached to nodes tell the "worth" of the node, i.e., they tell how useful the node has been in the past. In addition to being useful in searches, these flags tell the effector which nodes may be forgotten when the machine runs short on storage space.

The effector continually up-dates value-type flags to "reward" the nodes which are useful (by raising the number on the value-type flag attached to the node) and bring "closer together" groups of nodes which have been useful in combination with one another (by changing the numbers on the value-type flags attached to the lines on paths connecting the nodes to one another). Of course the act of completing the desired constructions provides new lines which can have flags whose values, in effect, draw together the nodes involved in the constructions.

By construction of new nodes a useful corpus of theorems is built up; and by change of numbers on value-type flags a certain amount of adaptation takes place.

1.6.2 Use of the Heuristics that are in the Net: An Example. A search for the nodes required for a complicated derivation would be hopeless if each possible combination of nearby previously rewarded nodes had to be individually and completely tested until a combination was found that worked. The effector needs a way to quickly reject, at least pro-

visionally, whole classes of possible combinations. Through the use of heuristics the effector can sometimes provisionally reject a combination of nodes quickly after examining only a few of the nodes, thus simultaneously rejecting all other combinations which use those few nodes.

Consider a simple example: Suppose the net is constructed as in Figure 4, except that the nodes below the broken line have not been constructed. Suppose that the effector wants to prove Theorem 5 and suspects that Rule 1 is the rule it wants to use. Now it is looking for two theorems to apply the rule to, and two parameter values. Its search will be simpler if Heuristic 1 (see below) is near Rule 1 in the net.

Heuristic 1: If γ is a theorem of the propositional calculus with \supset as its major connective and β as its consequent, then β is a theorem.

(Note that this is related to and "simpler" than Rule 1.) It is easier to apply Heuristic 1 than it is to apply Rule 1, because one need only look for one theorem, not two.

In Figure 5, the solid lines indicate the derivation of Theorem 5 via Heuristic 1. Note that heuristics may be told from real rules of inference by the fact that their tag-type flags have an H instead of a T on them. Also, Theorem 5 has an H on its flag because it was derived via a heuristic, rather than via a rule of inference, and hence there is no guarantee that it is indeed a theorem. It is essential to remember that Heuristic 1 is near Rule 1 in the net but far from Rule 2. (This is indicated by the dotted arrow in the figure.) (In more complicated cases there is a whole class of rules close to Heuristic 1 and we must try them all until we find one that works.) Thus the construction of the solid line derivation node has moved Theorem 4 and Theorem 5 closer to Rule 1, which

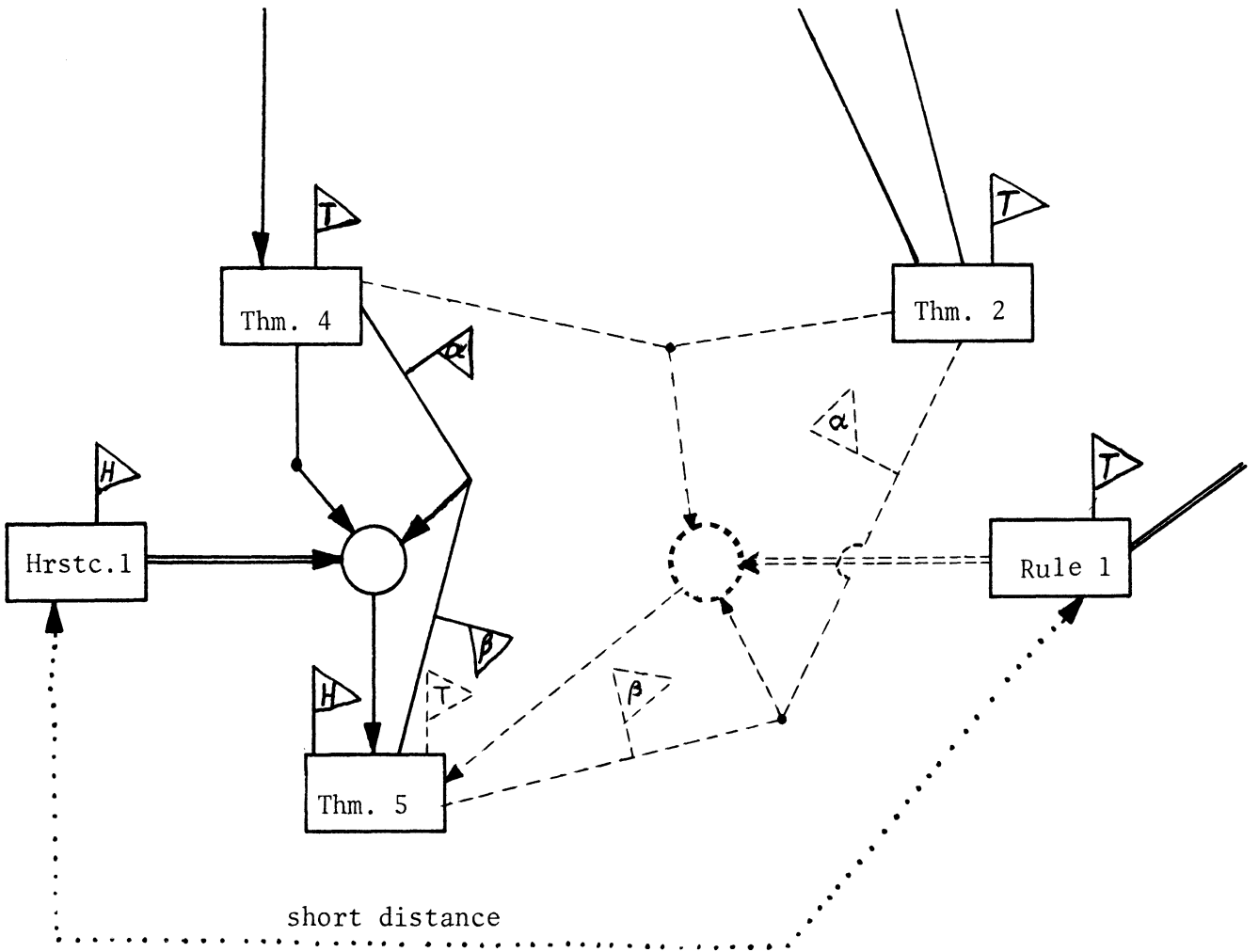


Figure 5.

is just what is needed to help the search for the proper nodes to permit the application for Rule 1. (The larger the net, the more important this moving becomes.) Thus the only really difficult search remaining is the search for Theorem 2.

With each theorem examined in this search, the effector attempts to apply Rule 1 to it and to Theorem 4 so as to yield Theorem 5. Of course, there are several ways this application can be made (depending on parameter value choices etc.), but this problem is small compared to the problem of finding Theorem 2 somewhere in the net. When finally the effector finds that Theorem 4 and Theorem 2 satisfy the conditions required by Rule 1,

then the broken line derivation node in Figure 5 is constructed and the T flag is added to the node containing Theorem 5 because Theorem 5 has now been legitimately proved via a real rule of inference.

Thus the search for the proper pair of theorems has been broken into two stages. Instead of checking each pair (μ, ν) of theorems in the net, the effector first searches for the proper μ , until it finds one that satisfies the heuristic. When such a μ is found, a construction is made which moves it close to the rule to be used. Then when the effector searches for a (μ, ν) which satisfies the rule, it is almost certain to pick the μ which has been moved, thus effectively rejecting all pairs using a different μ . Of course there is no guarantee that the effector has moved the right μ . (The idea behind heuristics is only that they help the effector make good guesses.) In the example above the effector wanted Theorem 4 to be μ , but Theorem 3 would have satisfied the heuristic just as well. In that case Theorem 3 would have been moved, the effector would have gone on a wild goose chase, and eventually given up and returned to the heuristic to look for something which worked better than Theorem 3. Thus the effector might have picked Theorem 3 by mistake, but at least it would not have picked Theorem 1 or Theorem 2 because they do not satisfy the heuristic. Thus in using the heuristic the effector provisionally rejects all (μ, ν) combinations in which μ is either Theorem 1 or Theorem 2.

1.6.3 Refining a Proof.

1.6.3.1 The General Scheme--*refineproof*. We have just seen a simple example of replacing an illegitimate "proof" which employs a heuristic with a legitimate proof which employs a rule of inference. The proof employing the heuristic is really a sort of proof "outline", and the replacement process is called refining the proof.

The basic job of the effector is the job of refining proofs. A proof outline, the steps of which are steps using heuristics, is gradually refined until we have a completed legitimate proof using only rules of inference. The above discussion has given a simple example of this process of refining a proof, where the part of the proof being refined was only one step long. More interesting examples will be given in the implementation section.

In more complicated examples the effector does not replace the heuristic step directly with a rule of inference step, but rather with another heuristic step which is more detailed. This is replaced by yet another heuristic step, and then another until finally it is replaced by a rule of inference. With each replacement, more conditions are checked, and more of the required nodes are brought in so that at each stage the effector checks to see that it is on the right track. (If it is not, it goes back a step and refines differently.)

In the above refining process the single step of the proof "outline" becomes a single step of the final proof. In general, however, this will not be the case. In many cases a stage in the refining process will consist in replacing a single step (i.e., single derivation node) with two (or, rarely, more) steps (i.e., two derivation nodes). Thus what was at first a single step in the proof "outline" can end up as a whole complicated derivation consisting of many steps in the final proof. We will give some examples in the implementation section. The actual procedure for constructing a proof is to begin with a one-step proof "outline" and then refine it stage by stage.

The heart of the effector is a recursively written program called *refineproof*. Its job is to completely refine a single step of a proof "outline". This is how it works. Suppose *refineproof* is presented with

a single step of a proof outline. If the step employs a rule of inference then *refineproof* is finished. If the step employs a heuristic then *refineproof* tries to replace this step with a new, more refined step or steps as discussed above. If this is successful, then *refineproof* has completed the first stage in the process of refining the single step. *refineproof* then calls itself recursively and applies itself to the new proof step, or successively to each of the steps if there is more than one. If the result of these applications is a complete legitimate proof, then *refineproof* is finished. If a complete legitimate proof is not the result, then the first stage in the process of refining the original single step was probably performed incorrectly. *refineproof* goes back to the original single step proof and tries again, this time replacing the single step with a step (or steps) different from the step (or steps) used in the previous unsuccessful attempt.

Note that after each stage in the refining process, all pertinent formulae in the net, while they may not yet be proved, may be said to be "semi-proved" in the sense that the node in which they stand is at the end of a proof "outline" the steps of which may employ heuristics as well as rules of inference.

The procedure, then, for constructing a proof is to begin with a one step proof "outline" and simply present this step to *refineproof*. It matters very little what the original one-step proof outline is; it is the connections between the heuristic used and the other heuristics and rules in the net that governs the effectiveness of the *refineproof* procedure and hence the usefulness of the heuristic used. The following heuristic is as good as any for use in a one-step proof outline which is to be refined.

Heuristic 2: If α is a well-formed formula of the propositional calculus,

then α is a theorem.

1.6.3.2 *prove*--and the Problem of Loose Ends. Consider a situation in which *refineproof* is faced with the problem of replacing a single step of a proof outline with two steps (i.e., of replacing one derivation node with two). For this task it calls on a program called *prove*. Remember that if the heuristics being employed are good ones, most of the nodes which *prove* needs in order to construct the new two-step derivation are nearby. Suppose the step to be replaced was a step which derived formula α . *prove* now tries to construct a new and more detailed proof outline from nearby nodes such that the formula derived is α . In doing this it builds backwards from α . That is, it tries to build the "second" step of the new two-step outline first. If it has selected a prospective rule and parameter values for the second step, it can tell easily (and recursively) what formulae need be attached to the antecedent node to make the derivation work. (These formulae may not yet be in the net, but *prove* can construct them.) *prove* constructs the required formulae and attaches them to the antecedent node. Now these formulae may not yet have been proved. In fact, as the net stands, they may not even be "semi-proved" in the sense discussed above. They are "loose ends"; they are formulae from which something has been derived but which have not themselves been derived from anything. *prove* then tries to construct a proof outline of each of these in turn.

Whenever a formula at one of these "loose end" nodes is also present at a nearby node which is not a "loose end" node, then the solution is trivial; *prove* simply merges the two nodes which contain the same formula, and the "loose end" disappears. This solution, however, will not eliminate all "loose ends". For example, one which cannot be so eliminated

contains the formula β , which is supposed to eventually be derived in the first step of the two-step outline that *prove* is constructing. *prove* tries to construct a derivation of β in the same way it constructed the derivation of α , thus producing a new set of "loose ends". If *prove* is lucky, these "loose ends" will all contain formulae present in nearby nodes which are not "loose ends". Then *prove* gets rid of "loose ends" by merging nodes containing identical formulae, as we discussed above. At last there are no more "loose ends". Then and only then is control returned to *refineproof* which tries to further refine each step of the new two-step outline.

Notice that no attempt is made to refine the second step before the first step has been constructed and all "loose ends" have been tied up. It is important that this not be done. The heuristic used in the second step is supposed to be testing whether or not the machine is on the right track. Remember that the formulae at the "loose ends" were manufactured specifically to allow the heuristic to work; thus the most important part of the test is deciding whether or not the "loose ends" can be tied up. If they can not, then any "successful" job of refining step two will leave the machine with a derivation which, while it looks fine by itself, is a derivation that should be rejected since it does not fit into the overall proof. Furthermore, this derivation is a member of the class of derivations that the heuristic used in Step 2 was specifically designed to reject. If the "loose ends" cannot be tied up, *prove* must reject the Step 2 that it has constructed and try to construct a different one.

Thus it is important that *prove* keep careful track of which derivations still contain "loose ends" and which do not. For this purpose it uses the tag-type flags which contain an H. Any node which is part of a

complete proof "outline" (with no "loose ends") in the net has an H flag attached to it. Other nodes do not. For example, no "loose end" has an H flag. Thus *refineproof* only attempts to refine steps in which all nodes have H flags. After each successful stage in the refining process (even if *refineproof* is ultimately on the wrong track) all pertinent formulae have H flags.

1.6.4 Saving Examples of Derivations Employing Heuristics. After a proof outline has been successfully refined to a legitimate proof, the outline and the partially refined intermediate steps in the refining process can be forgotten (i.e., eliminated from the net), but frequently it is better not to. For one thing, each of those derivation steps helps connect the heuristic used with the heuristics or rules of inference used in the more refined version of that derivation step. For example, in Figure 5, the derivation in solid lines helps make a closer connection, via Theorem 5, between Heuristic 1 and Rule 1. Thus the successful partnership of Heuristic 1 and Rule 1 is re-enforced by remembering a case in which the partnership was useful.

There is another reason for saving previously successful proof "outline" steps: they give information about the exact way in which the heuristic used is related to the rules which replaced it when the "outline" was refined.

To see how useful this information can be, consider the example shown in Figure 5. This example shows some details about a common use of Heuristic 1 (in fact, practically the only use of this heuristic). A single-step derivation using Heuristic 1 is usually replaced by a single-step derivation of the same theorem, this one using Rule 1. Usually the Heuristic 1 derivation and the Rule 1 derivation are related as follows:

(A) The parameter value for the parameter β is the same in the two derivations.

(B) The parameter value for γ in the Heuristic 1 derivation is one of two formula nodes attached to the antecedent node in the Rule 1 derivation.

(C) The other formula (call it ξ) node attached to the antecedent node of the Rule derivation is also the parameter value node for the parameter α in the Rule 1 derivation. This formula node is, in general, not part of the Heuristic 1 derivation.

Given any Heuristic 1 derivation, and with the above facts at its disposal, *prove* can almost construct directly the Rule 1 derivation to replace it. The only thing needed is the formula node ξ for which *prove* must search. *prove* examines the appropriate formula nodes in turn, testing each one to see whether or not it would make a legitimate ξ . When it has found one that would (Theorem 2 in the Figure 5 example) it constructs the appropriate derivation. The test of a prospective ξ will be efficient and direct if use is made of facts (A), (B), and (C) above.

If *prove* does not have facts (A), (B), and (C) at its disposal, the testing of a prospective ξ will be much more tedious. For each prospective ξ , the ξ will have to be combined in various ways with the formula nodes in the Heuristic 1 derivation. Without facts (A), (B), and (C), *prove* would have to try, in the example of Figure 5, ridiculous combinations such as

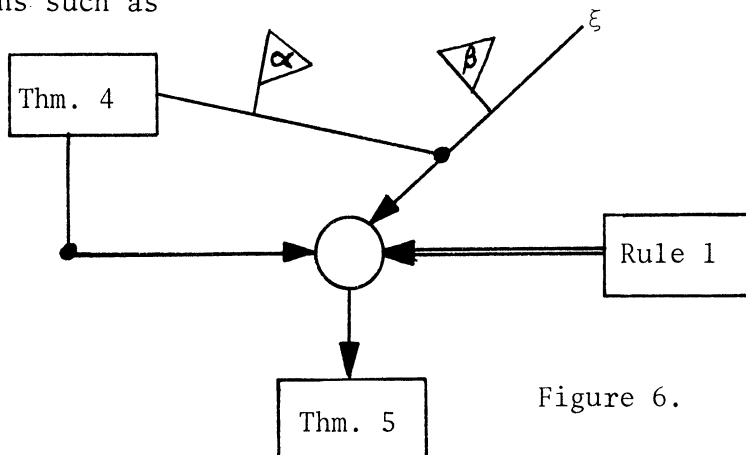


Figure 6.

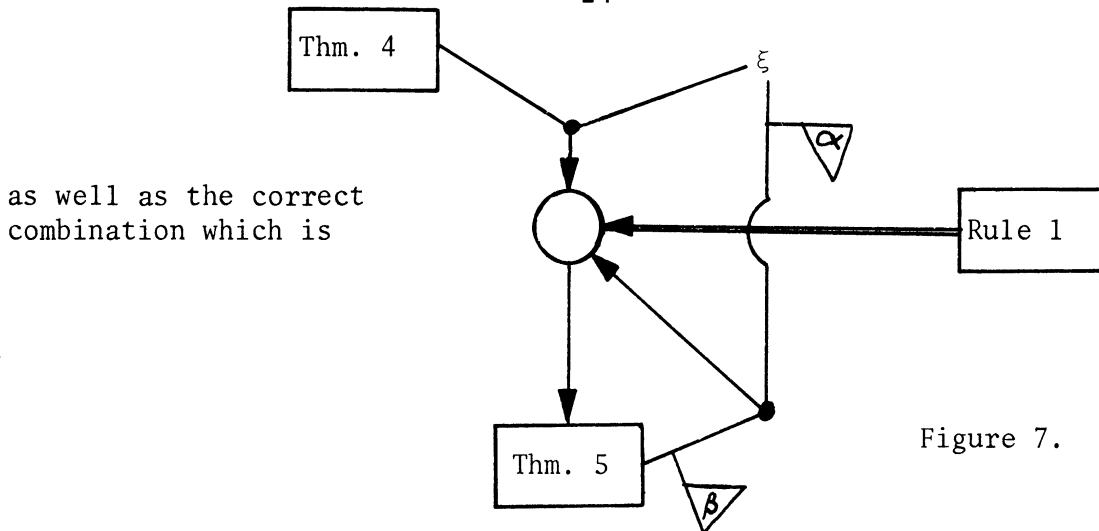


Figure 7.

as well as the correct combination which is

If *prove* has possession of facts (A), (B), and (C), it can limit combinations tested to the one shown in Figure 7.

How does *prove* obtain these facts? By reference to previously proved problems in which a Heuristic 1 derivation was replaced. If proof outlines and partially refined intermediate steps of previous problems are retained in the net, then attached to the Heuristic 1 nodes are various derivation nodes of Heuristic 1 derivations in previous problems. *prove* selects one of these. Suppose from some previous problem's proof outline, it selects a Heuristic 1 derivation of a formula η . Now when that proof outline was refined, the Heuristic 1 derivation of η was replaced (though the Heuristic 1 derivation was retained in the net) with a Rule 1 derivation of η , and this derivation is still in the net. Thus η is surrounded by exactly the same structure that Theorem 5 is surrounded by in Figure 5.

In doing a new problem, *prove* simply tries to mimic the structure of this previous model problem. Thus it automatically restricts its attention to derivations of the form of Figure 7, because that is the form of the derivation it is mimicking; that is what worked before. Model problems successfully mimicked are rewarded so that they become more likely to be used next time. A large part of the machine's adaptation takes place in this way.

Of course such mimicking does not always work, as, for example, when a heuristic is being tried for the first time. In this case we resort to simply checking which nodes are near the heuristic, and trying to combine them as discussed earlier. In any case, it is important that the rule or rules with which the heuristic is to be replaced be close to the heuristic in the net (since this is how candidates for replacement are selected). When the mimicking technique works, the very lines which connect the heuristic with the rule or rules (thus causing them to be close together) give the relationship between the structure of the heuristic derivations and the structure of their replacements.

The behavior of *prove* will be discussed more explicitly in the implementation section.

Thus the net contains not only completed proofs, but successful partly refined proof outlines. The role of the heuristics in achieving the solution is thus preserved.

1.7 Generation of Heuristics

In Section 1.3 we mentioned that once the machine proves a theorem which relates structure of LISP programs to their performance, the machine can often use the theorem to help generate a new heuristic directly from a rule of inference. The generation of a new heuristic is, in form, much like the derivation of a new theorem. In each case, a derivation node is added to the memory net, and from it an arrow points to a new formula node containing the new formula or the new heuristic (written in the object language of S) as the case may be.

For example, suppose the machine were to generate Heuristic 1 directly from Rule 1. The record of the generation would look (in part) as shown in Figure 8. (In Section 3.8 we shall discuss the nodes not shown here.) This record looks just like a step in a proof outline, and,

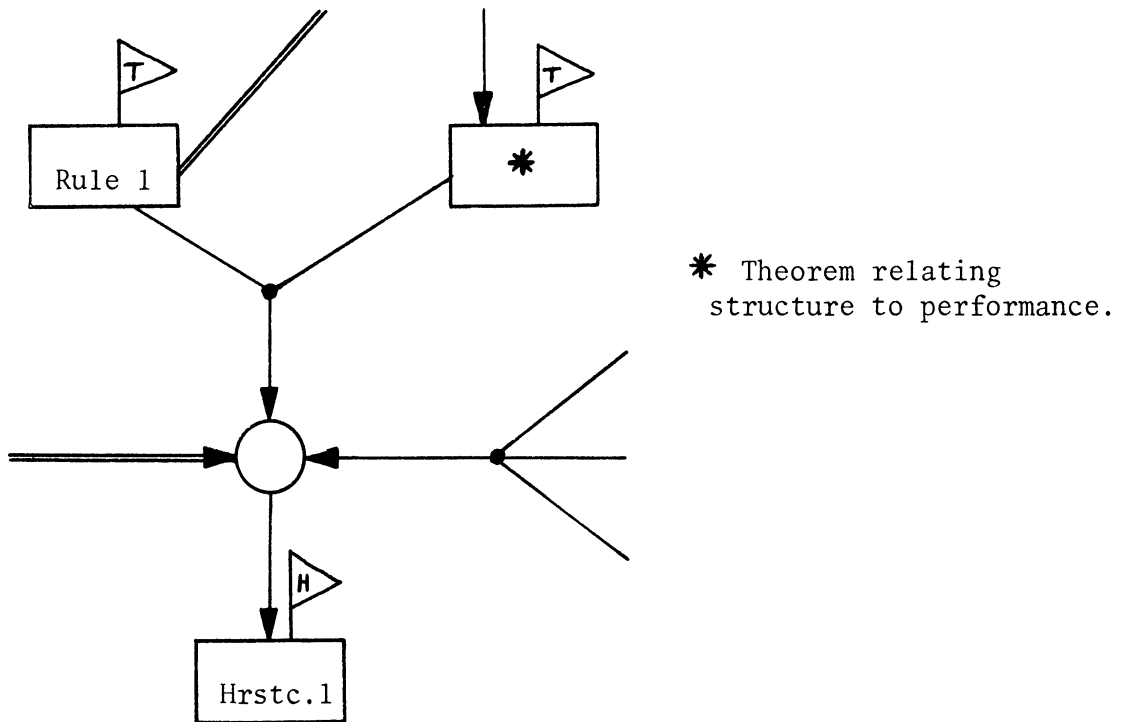


Figure 8.

in fact, as we shall point out in Section 3.8, that is what it is. (We need the formalism of Sections 2.1 to 3.7 to see why it is genuinely a step in a proof outline.)

Now suppose Heuristic 1 is employed in a step of a proof outline as it was in the solid line construction of Theorem 5. The effector wishes to replace the derivation employing Heuristic-1 with a derivation employing a rule, a rule which is nearby. What rule is nearby? Since Heuristic 1 has just been generated and has never been used before, it is not connected to any rule by means of a sample problem which has been saved in the manner discussed in Section 1.6.4. Its only close connection to a rule is its connection to Rule 1 via the derivation shown in Figure 8. This connection is what was indicated cursorily by the dotted line in Figure 5. This explains the connection via the dotted line in Figure 5, the connection that was, in that example, so important to the effector in deciding which rule to select in trying to replace the heuristic derivation.

The situation we have described is rather typical. When faced with the problem of replacing a derivation employing a brand new heuristic, the effector, not having any previous sample problem to look at, will look at the nearby rule and this will be the rule from which the heuristic was generated. If our scheme for generating heuristics from rules is at all reasonable, this is just where the effector should look.

1.8 Questions Beyond the Scope of the Paper

The implementation section will show the existence of at least one interesting adaptive theorem proving machine which treats heuristics in a general way. Thus it will show that the class of such interesting machines is non-empty. Eventually one would like to discuss performance of members of the class in various problem environments (i.e., various sequences of problems presented, and various rewards given the machine for correct answers--see Section 1.4.). One would compare the performance of various machines in representative environments. From this would arise a meaningful classification of environments based on behavior of various machines in those environments. For any two machines, there will be environments in which the first adapts more rapidly than the second and others in which the second adapts more rapidly than the first.

Any measure of adaptability is, then, environment dependent, and any meaningful classification of environments must depend on adaptability of various machines when faced with them. Such a theory would allow us to compare machines with each other. The value of such a theory would be that it would in fact be a general theory of heuristics. It would allow us to compare one machine with another on the basis of heuristic generation methods (structure) and adaptability in various environments (performance).

The question as to whether any machines in the class are interesting is, of course, a subjective one. An answer would require establishing the adaptability of a member of the class not in the various categories of environments we discussed above, but in a large number of sample environments chosen for their intuitive interest (a characteristic not considered in the above method of classifying environments). This would have to wait for the actual construction of the machine which we claim to be interesting.

We have taken a small step in this direction. We have shown in the implementation section that there is a member of our class of machines which performs as well as Newell, Shaw, and Simon's machine in the environment used by Newell, Shaw, and Simon. (This is generally recognized to be an intuitively interesting sample environment.) We have further shown that it is easy to incorporate into this machine, in a very general way, the improvements which naturally come to mind. (Many were suggested, but not all tested, by Newell, Shaw, and Simon.) In most cases, these improvements are implicitly in our machine already since their incorporation means not the construction of a new part of the machine, but the addition of a few heuristics to the initial memory net.

2. THE FORMAL SYSTEM

2.1 Basic Structure of the System

2.1.1 Overview of System

2.1.1.1 Plan of Discussion. We now begin the description of the formal system which is the basis of the adaptive theorem prover.

The formal system has the following parts:

- A. A set of symbols called the alphabet.
- B. A set of formation conventions by which the members of the alphabet may be combined to form expressions of various kinds, one kind being called formulae.
- C. A finite collection of formulae called axioms.
- D. A collection of rules of inference which when applied successively to the axioms generate an infinite set of formulae called the set of theorems.
- E. An infinite set of expressions called the set of well-formed expressions. This class is recursive relative to the set of theorems, and includes the set of theorems.
- F. An intended interpretation for the well-formed expressions such that the theorems become true statements. (When we say that a formula holds, we will mean that it is true in the intended interpretation.)

These parts will all be described formally in later sections. We begin here with an informal description.

We can think of the system as being obtained by a series of transformations on a finitely axiomatized first order formal arithmetic. Each transformation preserves consistency so that our system will be just as consistent as the original arithmetic. The formal arithmetic will be one with propositional variables, individual variables, predicate variables and individual function variables, as well as the binary predicate constant = interpreted as equality, the individual constant \emptyset interpreted as zero, the

unary individual function constant S interpreted as the successor function, the binary individual function constant $+$ interpreted as addition, and the binary individual function constant \cdot interpreted as multiplication.

We shall first discuss briefly the formal arithmetic and then discuss each transformation in turn. Each transformation is characterized by an addition or subtraction of symbols from the alphabet together with concomitant changes in formation conventions, axioms, rules, and the set of well-formed expressions to allow the new symbol to be incorporated into the language and to permit the intended interpretation we wish it to have.

In this Section (2.1.1) we shall discuss in turn each transformation by discussing the symbols added or subtracted and discussing their intended interpretation. We will indicate from time to time what the concomitant changes are in the formation conventions, axioms, rules, or set of well-formed expressions. We will not, however, give these concomitant changes in detail. We shall reserve such formal discussion for later sections where we shall present our formal system explicitly. The discussion in Section 2.1.1 should give the reader an overview of our system. The overview will provide a certain motivation for the notation to be described in the later formal discussion. The notation used there might be rather confusing otherwise.

As we mentioned, each transformation of the formal arithmetic is characterized by the addition or subtraction of certain symbols from the alphabet. The first transformation is characterized by addition of ι . The

second transformation is characterized by addition of λ etc., as indicated below.

<u>Transformation</u>	<u>Characterization</u>
transformation 1	add ι
transformation 2	add λ
transformation 3	add label, cond, and pcond
transformation 4	subtract + and \cdot
transformation 5	add * and nil
transformation 6	add Pv, Iv, Pfvb, Ifvb, newpv, newiv, newpfvb, and newifvb; subtract tv, I, Pf, and If
transformation 7	subtract S and \emptyset and add qu

When we have finished the last transformation we will have arrived at our system.

The language of our system is explicitly self-referential rather than being explicitly about numbers and only self-referential via a Gödel numbering as is the formal arithmetic.

2.1.1.2 Pertinent Properties of the Formal Arithmetic. We

begin with a formal arithmetic with plus and times. Since we will be changing the axioms anyway it is not crucial just which axiomatization we begin with as long as the axiom set is finite. In order for the axiom set to be finite, we employ predicate variables and individual function variables, with rules of substitution for these variables.

For example, we can use as logical axioms and rules of inference, the axiomatization of Church's system F_2^{1p} [p. 218-219, Church, 1956] with suitable modification of the alphabet, formation conventions, and rule of inference $*404_n$ to include individual function variables, and the various constants. (We regard $*404_n$ as a single rule.)

The additional axioms to take care of the various constants could be, following Mendelsohn [Mendelsohn, 1964]

for equality(using infix notation):

$$x = x$$

$$x = y \supset (P(x) \supset P(y)) \quad (P \text{ is a predicate variable})$$

for successor, the Peano axioms:

$$\sim(\emptyset = S(x))$$

$$S(x) = S(y) \supset x = y$$

$$P(\emptyset) \supset ((\forall (x) (P(x) \supset P(S(x)))) \supset \forall(x) P(x))$$

(where our $\forall(x)$ is Church's (x))

for addition.(using infix notation):

$$x + \emptyset = x$$

$$x + S(y) = S(x+y)$$

for multiplication (using infix notation):

$$x \cdot \emptyset = \emptyset$$

$$x \cdot S(y) = y + (x \cdot y)$$

As we mentioned, our system will use both individual function variables and predicate variables as well as propositional variables and individual variables. The propositional variables are: p, q, r, s, p, q, r, s, p,.... The individual variables are: x, y, z, u, v, w, x, y, z, u,.... Each of these letters, together with its numeral subscript, is regarded as a single symbol of our alphabet. The alphabet is thus infinite.

In the case of predicate variables, we shall use the subscript I to indicate the number of arguments. The I is a separate symbol of our alphabet.

Unary predicate variables: $P_I, Q_I, R_I, P_{1I}, Q_{1I}, R_{1I}, P_{2I}, \dots$

Binary predicate variables: $P_{I,I}, Q_{I,I}, R_{I,I}, P_{1I,I}, Q_{1I,I}, R_{1I,I}, P_{2I,I}, \dots$

Ternary predicate variables: $P_{I,I,I}, Q_{I,I,I}, R_{I,I,I}, P_{1I,I,I}, \dots$ etc.

Thus the ternary predicate variable $P_{I,I,I}$ is made up of four symbols: the so-called base symbol P to which has been added three subscript symbols, I 's. These predicate variables, then, are formed by adding subscript I 's to the predicate variable bases: $P, Q, R, P_1, Q_1, R_1, P_2, \dots$. Each predicate variable base is a symbol of the alphabet.

In this discussion we shall sometimes omit the subscript I 's when the predicate variable occurs within a formula and the number of its arguments is clear from context. Thus, in the above axioms we have written P in place of P_I . This is only an abbreviation for purposes of brevity. For example, the second axiom of equality is really $x = y \supset (P_I(x) \supset P_I(y))$ in spite of the fact that we have written, and shall continue to write, $x = y \supset (P(x) \supset P(y))$.

The notation for individual function variables is similar to the notation for predicate variables. The individual function variable bases are $f, g, h, f_1, g_1, h_1, f_2, \dots$. To these we add the subscript I 's to indicate arguments. For example, $f_{I,I,I}$ is a ternary individual function variable. Again, when the arguments are clear from context we shall sometimes omit the I 's.

The predicates differ from individual functions in that their range is the class of truth values rather than the class of individuals (in this case, numbers). In both the predicates and individual functions discussed above, the domain for each argument was the class of individuals. We can naturally extend the notion of predicate and individual function to allow a domain to be the class of truth values, or even the class of individual functions or predicates. Thus \supset names a binary predicate, each of whose arguments is to be a truth value. $P_{tv,tv}$ is a binary predicate variable ranging over such predicates. The use of the subscript tv (a separate

symbol of the language) in place of I indicates the truth value nature of the arguments. Thus we use I to indicate individual-type arguments and tv to indicate truth value-type arguments. We also use Pf to indicate predicate-type arguments and If to indicate individual function-type arguments.

Consider, for example, an individual function like the LISP function *maplist*. This is a function of two arguments. The second argument is to be an individual (in this case a so-called S-expression). The first argument, however, is to be an individual function of one argument. More specifically, it is to be an individual function whose single argument is to be an individual.

$f_{If_I, I}$ is an individual function variable ranging over individual functions of the *maplist* type. The second subscript is I, indicating that the second argument is to be an individual. The first subscript is If, indicating that the first argument is to be an individual function. Since the single argument of this function is to be an individual, the subscript If is itself subscripted with a single I. This subscripting of subscripts may be repeated any number of times, so that the subscripting on variables may become rather complicated. However, the substitution rules for these variables are quite straightforward and are rather obvious extensions of Church's rule 404_n; hence we shall not give the substitution rules here, but wait until Section 2.1.4.4 when we can give them in our final notation.

As we said above, we shall often abbreviate predicate variables and individual function variables by writing the predicate variable base or individual function variable base without subscripts whenever the needed arguments are clear from the context. For example, we can abbreviate

$$f_{tv, If_{Pf_I, I}}(P_{tv, I}(p, x), g_{Pf_I, I}) \quad \text{as} \quad f(P(p, x), g_{Pf_I, I})$$

Another useful abbreviation we shall employ is our abbreviation for the so-called numerals. A numeral is any one of the following sequence of expressions:

$\emptyset, S(\emptyset), S(S(\emptyset)), S(S(S(\emptyset))), \dots$ etc.

In our discussion, we shall often abbreviate \emptyset as $\bar{0}$, $S(\emptyset)$ as $\bar{1}$, $S(S(\emptyset))$ as $\bar{2}$, $S(S(S(\emptyset)))$ as $\bar{3}$, etc. If k is a natural number, we call \bar{k} a numeral. The \bar{k} 's are not new symbols of our language, merely abbreviations we will use in this discussion.

This abbreviation makes it easy to state our representability property.

If A_{x_0, \dots, x_n} is a formula with free individual variables x_0, \dots, x_n then let

$A_{\bar{k}_0, \dots, \bar{k}_n}$ stand for the formula derived from A_{x_0, \dots, x_n} by uniform replacement

of $\bar{k}_0, \dots, \bar{k}_n$ for the free occurrences of x_0, \dots, x_n . We say an $n+1$ -ary

relation on integers is weakly representable whenever there is a formula

A_{x_0, \dots, x_n} such that for any natural numbers k_0, \dots, k_n , the relation holds on

k_0, \dots, k_n if and only if $A_{\bar{k}_0, \dots, \bar{k}_n}$ is provable.

In formal arithmetic with plus and times, every recursively enumerable relation is weakly representable. [Mendelsohn, 1964]

2.1.1.3 Addition of 1 (representability of individual functions). The graph of every n -ary partial recursive individual function is an $n+1$ -ary recursively enumerable relation, and is hence weakly representable. If the partial recursive individual function is a polynomial function, then its graph is weakly representable by a formula of form $x_0 = \alpha$, where x_0 is the individual variable for which is to be substituted the name of the element of the function's range, and α is a term not containing a free

x_0 . When this is the case, I shall say that the term α mimics the individual function. For each partial recursive individual function, we would like to find a term which mimics the function. In formal arithmetic with plus and times, this can be done only for polynomial functions.

Let us add to the system of formal arithmetic, the new variable binder ι . If A_{x_0, \dots, x_n} is a formula with free variables x_0, \dots, x_n , then $\iota(x_0) A_{x_0, \dots, x_n}$ is a term which means intuitively: the unique value for x_0 such that A_{x_0, \dots, x_n} holds, when such a value exists. When none such exists, **the term** will have an undefined value.

Suppose we add axioms to the system to implement this meaning. Now given any n-ary partial recursive function, its graph is weakly representable by some formula A_{x_0, \dots, x_n} . Hence the formula $x_0 = \iota(x_0) A_{x_0, \dots, x_n}$ is true for values satisfying the graph. Whether this second formula weakly represents the graph depends on whether the formula is provable in the cases mentioned above where it is true. Proving the second formula for values satisfying the graph is harder than simply proving A_{x_0, \dots, x_n} for these values because we now have to prove the uniqueness of the x_0 's for each set of x_1, \dots, x_n 's in the domain of the function. Because of the incompleteness of formal arithmetic, then, there will be formulae A_{x_0, \dots, x_n} such that:

A_{x_0, \dots, x_n} weakly represents the graph of a partial recursive function ϕ , but the formula $x_0 = \iota(x_0) A_{x_0, \dots, x_n}$ does not weakly represent the graph of ϕ . The following however is true: Given a partial recursive function ϕ , there is a formula A_{x_0, \dots, x_n} such that both A_{x_0, \dots, x_n} and $x_0 = \iota(x_0) A_{x_0, \dots, x_n}$

weakly represent the graph of ϕ . (In most practical cases, if A_{x_0, \dots, x_n} weakly represents the graph of a partial recursive function ϕ , so does $x_0 = \lambda(x_0) A_{x_0, \dots, x_n}$.) Thus we can mimic any partial recursive function we want to.

2.1.1.4 Addition of λ (ordinary - names of functions

and relations). So far our system contains names for only a few individual functions (Eg. plus and times). We shall introduce names for all the partial recursive functions. We shall do this via the Church λ notation. If

B_{x_1, \dots, x_n} is a term which mimics an individual function, then the expression

$(\lambda, (x_1, \dots, x_n), B_{x_1, \dots, x_n})$ is a name for the function. (Similarly, if

A_{x_0, \dots, x_n} is a formula which represents a recursively enumerable Relation,

then the expression $(\lambda, (x_0, \dots, x_n), A_{x_0, \dots, x_n})$ is a name for that

relation.) We shall add axioms to the system to implement this meaning in the proof structure. Since for any partial recursive function there is a term which mimics it (and for any recursively enumerable relation there is a formula which weakly represents it) we have, in the above way, provided an expression naming each partial recursive function and recursively enumerable relation. We shall call such expressions **ordinary-names** of the functions and relations.

2.1.1.5 Addition of label, cond, and pcond

(algorithmic-names of functions and relations.) Our interest in such functions

and relations stems from the fact that they possess algorithmic evaluation procedures. When we can find such an algorithm we will write an expression which describes the algorithm. This expression will be in a LISP-like notation (my LISP is like LISP 1.5 [McCarthy 1962] except that

I have made some minor changes in notation and one significant change in the evaluation procedure) and will be called an algorithmic name for the function. The structure of such an expression reflects the particular algorithm it describes. The notation of our system is so close to LISP notation that any polynomial function has an ordinary-name which is also an algorithmic-name. The only added symbols needed to complete the LISP repertoire for writing functions defined over non-negative integers are the symbols label, cond, and pcond. (pcond is an alternative spelling for cond. It is used in certain cases described in a later section.) label is the symbol which is used to indicate explicit definition by recursion. cond and pcond are especially useful in such definitions. Our usage is almost identical to LISP usage.

We will not describe the usage in detail here, but the reader unfamiliar with LISP usage may benefit from the following example: The expression $(\text{cond}, (\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n))$ which we abbreviate to $[\alpha_1 \rightarrow \beta_1; \alpha_2 \rightarrow \beta_2; \dots; \alpha_n \rightarrow \beta_n]$ has the same value (for a particular set of values for its variables) as does β_j , where j is the smallest number such that α_j has value true. We use label, combined with cond to write an algorithmic-name such as this one for the primitive recursive function *factorial*: $(\text{label}, f, (\lambda, (x), [x = \emptyset \rightarrow \bar{1}; \oplus \rightarrow x \cdot f(x - \bar{1})]))$. For purposes of illustration we have used here a subtraction symbol which we have not yet defined. Note that this expression reflects the usual primitive recursive definition of *factorial*. The dummy function variable f which follows the label is to be regarded inside the definition as naming the function we are trying to define. One can use label in recursions which are not primitive recursions.

We need to do more than merely apply algorithmic names, LISP fashion. We need to put rules in the system which allow us to process and modify these algorithmic names according to various rules of inference so that we can compare them with each other and with ordinary names. An example of such modification would be: substitution of argument terms into the matrix of a λ expression according to one of the rules of inference (analogous to a stage of LISP application of a function). An example of comparison would be: if ϕ is an ordinary-name and ψ is an algorithmic-name, then we will want to prove $\phi(x) = \psi(x)$ if we can, so that we can then substitute ψ for ϕ in theorems, thus turning these theorems into statements about the performance of the LISP program ψ .

Unfortunately if a function or relation is not a total function or relation, it may have an algorithmic name which, when processed in the ways we want to process algorithmic names, would produce a contradictory statement and ruin the consistency of the system. For example, consider the function name (label, f, (λ , (x) , (f(x) + $\bar{1}$))) which we abbreviate as ϕ . Then $\phi(y)$ (by the recursive definition of ϕ) evaluates to $\phi(y) + \bar{1}$. We can even prove $\phi(y) = \phi(y) + \bar{1}$ from $\phi(y) = \phi(y)$ by the process of partial evaluation of the right hand side. Hence $\phi(y) \neq \phi(y)$ is provable and the consistency is ruined. We want to declare such "contradictory" algorithmic-names illegal. It turns out, luckily, that we have no need for these contradictory algorithmic names, because every partial recursive function has at least one non-contradictory algorithmic name.

How do we set up the formalism so the contradictory names are never used? All contradictory names contain label, but beyond this they have no easily recognizable structural characteristic. We don't want to limit ourselves to only complete functions. For example, we can easily allow all ordinary names to be used. They are all non-contradictory.

Here is what we do. We carefully control which label-containing names are used. We start with a finite set of such names called the initial set of algorithmic names. (These will be the names occurring in the axioms.) We then write the rules of inference so that no new label is introduced unless either: (1) It is part of a name which has appeared previously in a theorem; or (2) it is part of a new name which has been generated from an already occurring name according to a special procedure which guarantees that the new name will be non-contradictory. This is an oversimplified description of a scheme which will be described in detail in a later section.

When a theorem is proved which contains a function name not occurring in any previous theorem (whether or not the proof employs (2) above to introduce a new label) then we say that that function name is generated.

A function name which can be generated, is called generatable. (The initial set of algorithmic names is thought of as already generated and hence trivially generatable.)

We have now arrived at the following: Given any n-ary partial recursive function with ordinary name ϕ , we can write a LISP program ψ which calculates its value. ψ is then an algorithmic name of the function. We can even write the LISP program such that ψ is a non-contradictory algorithmic name. Then

$\forall(x_1) \dots \forall(x_n) (\phi(x_1, \dots, x_n) = \psi(x_1, \dots, x_n))$ is true, but ψ might not be generatable and even if it is, the above formula might not be provable.

Footnote: Sections 4.10.6 and 4.10.7 will illustrate situations where ψ is generatable, but not generatable from ϕ , and where

$\forall(x_1) \dots \forall(x_n) (\phi(x_1, \dots, x_n) = \psi(x_1, \dots, x_n))$ is probably not provable.

However by picking the proper axiom set we can ensure that every partial recursive function has a generatable algorithmic name.

Footnote: we need only be sure that the class of functions which have generatable algorithmic names is closed under primitive recursion and the μ operator. Section 4.10.7 will indicate how to show closure under the μ operator.

Similarly, given any n-ary recursively enumerable relation with ordinary-name ϕ , we can write a LISP program ψ such that ψ is a non-contradictory algorithmic name and $\forall(x_1) \dots \forall(x_n) (\phi(x_1, \dots, x_n) \equiv \psi(x_1, \dots, x_n))$ is true.

Again Ψ might not be generatable and even if it is, the above formula might not be provable. However, by picking the proper axiom set we can ensure that every recursively enumerable relation has a generatable algorithmic name.

When the above equivalence statements have been proved, we can use the substitutativity of $=$ and \equiv (subject to the proper restrictions made explicit in our rules of inference) to change theorems using ordinary names to theorems using algorithmic names. (Theorems using algorithmic names can often be proved more easily without use of the above equivalence statements.) Thus we can prove a class of true statements about LISP programs. Of course, we cannot prove all true statements because the formal arithmetic we began with was incomplete, and the incompleteness is retained through each of our transformations of the system. (We shall prove the incompleteness of the final system.) Thus incompleteness arises from the self-describing capability of all these systems, a capability of which we shall make explicit use in our final system.

2.1.1.6 Elimination of + and · . Note that it is possible for the names in the initial set of algorithmic names to be names of functions which have no ordinary names. For the system as described so far, the initial set of algorithmic names is empty.

Let us write algorithmic names for plus and times such that the symbols + and · are not used in the names. (The symbol S is used. The names are simply the LISP programs implementing the primitive recursive definitions of plus and times in terms of successor!) Let us abbreviate these two names as \oplus and \odot . (These are not new symbols, just abbreviations we will use here to represent the algorithmic names.) These names are non-contradictory since they are names of complete functions.

Then let us put \oplus and \circ in the initial set of algorithmic names. (This could be done by adding the axiom $\exists(x) (x = ((y \circ z) \oplus w))$. In this section (2.1.1) we will mention many algorithmic names which we want in the initial set. We shall wait until we have a long list of such names before we discuss which axioms are to be added to include the names in the initial set.)

It is easy to see, from the meanings of λ and label discussed above and expressed explicitly by our rules of inference, that if \oplus and \circ are in the set of initial algorithmic names, then for each axiom A_{+} using $+$ and \cdot , and hence for each theorem A_{+} using $+$ and \cdot , there is a theorem A_{\oplus} formed from A_{+} by uniform replacement of \oplus for $+$ and \circ for \cdot . Furthermore, none of the formulae in the proof of A_{\oplus} contain $+$ or \cdot . Conversely, for any theorem A_{\oplus} using \oplus and \circ , the formula A_{+} is also a theorem.

Let us eliminate the symbols $+$ and \cdot from the language, and all the formulae which use them. We thus remove the axioms which defined $+$ and \cdot . Now any relation which was weakly representable by a formula using $+$ and \cdot is still weakly representable by a new formula obtained from the old by replacement of $+$ by \oplus and \cdot by \circ . Thus any function or relation which had a name before still has a name. We have eliminated $+$ and \cdot from the language without destroying the representability of any function or relation. The only individual function constant symbol remaining in the language is S .

2.1.1.7 An Operation on S-expressions: Addition of $*$ and nil . Let us consider a Gödel numbering of the expressions in our language.

We first state what we mean by an S-expression (as in LISP): (a) Each symbol (or atom) in our alphabet is an S-expression. (b) If α and β are S-expressions, so is $(\alpha \ . \ \beta)$, which is called the "cons" of α and β .

If γ is the cons of α and β then α is called the "car" of γ and β is called the "cdr" of γ . (Note: in this scheme, the parentheses and dot are

not called numbers of the alphabet, but are introduced into S-expressions as part of the process of concatenating numbers of the alphabet.)

By writing our syntax rules carefully (see Section 2.1.2.2.), we can arrange matters so that all the expressions in our language are S-expressions, though, of course, not all S-expressions are expressions of our language. (Then the expressions we are writing in this text may be regarded as abbreviations of the appropriate S-expression. The abbreviation conventions are specified in Section 2.1.2.) In order to make it easier to write our syntax rules, we add the symbol nil to our alphabet. This symbol will be used as a sort of delimiter in constructing the expressions of our language. Its precise use is explained in Section 2.1.2.2.

Counting variables, our alphabet is countably infinite. Let us order the alphabet (excepting nil) in some appropriate order and let a symbol's index be its number in this ordering. Now let us Gödel number the S-expressions. To be specific, we could do it as follows. Suppose γ is an S-expression: if γ is nil then its Gödel number is zero;

if γ is any other atom (other than nil) with index n then its Gödel number is $2n - 1$;

if γ is the cons of α and β and if n is the Gödel number of α and m is the Gödel number of β , then the Gödel number of γ is $(2n + 1)2^{m+1}$.

The important point is that for each number, n , there is an S-expression with Gödel number n . Let us write the abbreviation $\tilde{\alpha}$ for \bar{n} whenever n is the Gödel number of α .

Now, via an algorithmic name for the exponential function, we can easily write an algorithmic name (which we abbreviate as \otimes) for the function cons. That is, if α has Gödel number k , β has Gödel number j , and the cons of α and β has Gödel number n , then $\bar{n} = (\bar{k} \otimes \bar{j})$ is provable. Or equivalently, $\widetilde{(\alpha, \beta)} = (\tilde{\alpha} \otimes \tilde{\beta})$ is provable. Let us put \otimes in the initial

set of algorithmic names.

Now let us add the binary individual function symbol $*$ to the language. It is defined by the new axiom $x * y = x \circledast y$. $*$ is thus a symbol for the function cons in the same sense that \circledast is an algorithmic name for the function cons.

It is trivial now to write ordinary names (using ι) for the functions car and cdr. (car and cdr of an atom will be the atom itself) We will abbreviate these names as a and d respectively. Thus, whenever, as before, $\bar{n} = (\bar{k} * \bar{j})$ is a theorem, so is $\bar{k} = a(\bar{n})$ and $\bar{j} = d(\bar{n})$. The names a and d contain the symbol $*$, but do not contain the symbol S.

2.1.1.8 Addition of Pv, Iv, Pfvb, Ifvb, newpv, newiv, newpfvb, and newifvb; subtraction of tv, I, Pf, and If.

We would next like to eliminate S from the language. We eliminated $+$ and \cdot by re-defining them in terms of S. Now that we have introduced $*$ we would like to eliminate S by re-defining it in terms of $*$. Unfortunately we are foiled by the fact that our alphabet is infinite, because of the four infinite sets of variables and variable bases included in it. As we mentioned in Section 2.1.1.2, these sets are:

1. The propositional variables $\{p, q, r, s, p_i, q_i, \dots\}$
2. The individual variables $\{x, y, z, u, v, w, x_i, y_i, \dots\}$
3. Predicate function variable bases $\{P, Q, R, P_i, Q_i, \dots\}$
4. Individual function variable bases $\{f, g, g, f_i, \dots\}$

If we eliminate S we will have no way of handling these sets. In order to handle these infinite sets and the Gödel numbers of their members, we need predicates: Pv, Iv, Pfvb, and Ifvb true respectively on Gödel numbers of members of the above four sets.

We also need individual functions: *newpv*, *newiv*, *newpfvb*, *newifvb*, with the following properties. For any S-expressions, α and β :

if *newpv* ($\tilde{\alpha}$) = $\tilde{\beta}$ holds then β is a propositional variable not in α ;

if *newiv* ($\tilde{\alpha}$) = $\tilde{\beta}$ holds then β is an individual variable not in α ;

if *newpfvb* ($\tilde{\alpha}$) = $\tilde{\beta}$ holds then β is a predicate function variable base not in α ;

if *newifvb* ($\tilde{\alpha}$) = $\tilde{\beta}$ holds then β is an individual function variable base not in α .

We add these predicates and functions in the same way we added *. We first define algorithmic names for the four predicates and four individual functions. We then introduce eight new symbols to be equivalent to these. We proceed as follows. It is easy to use S, \oplus , and \circ to define the following algorithmic names.

Abbreviation for algorithmic Name	Predicate Named
<i>Pv</i>	True on Gödel numbers of propositional variables
<i>Iv</i>	True on Gödel numbers of individual variables
<i>Pfvb</i>	True on Gödel numbers of predicate function variable bases
<i>Ifvb</i>	True on Gödel numbers of individual function variable bases

Similarly, we can define algorithmic names abbreviated *newpv*, *newiv*, *newpfvb*, and *newifvb*. The individual functions they name can be described as follows.

(Remember that we have an ordering of variables by their indices.)

newpv ($\tilde{\alpha}$) = $\tilde{\beta}$ holds iff β is the first propositional variable of index $>$ those in the S-expression α .

newiv ($\tilde{\alpha}$) = $\tilde{\beta}$ holds iff β is the first individual variable of index $>$ those in the S-expression α .

newpfvb ($\tilde{\alpha}$) = $\tilde{\beta}$ holds iff β is the first predicate function variable base of index $>$ those in the S-expression α .

newifvb ($\tilde{\alpha}$) = $\tilde{\beta}$ holds iff β is the first individual function variable base of index $>$ those in the S-expression α .

Footnote: An example of how this can be done is seen below. We use here notation and abbreviations which are explained in Section 2.1.2.

$$\text{maxx}(x, u, y, v, f) =: [u = y \rightarrow y ; \oplus \rightarrow f(x, S(u), y, v)]$$

$$\text{maxy}(x, u, y, v) =: [v = x \rightarrow x ; \oplus \rightarrow \text{maxx}(x, u, y, S(v), \text{maxy})]$$

(maxy causes no problems even though it is not complete.)

$$\text{max}(x, y) =: \text{maxx}(x, x, y, y, \text{maxy})$$

$$\text{evenn}(x, y) =: [x = y \rightarrow \oplus ; x = S(y) \rightarrow \oplus ; \oplus \rightarrow \text{evenn}(x, S(S(y)))]$$

(similarly this causes no trouble)

$$\text{even}(x) =: \text{evenn}(x, \emptyset)$$

$$\text{maxpv}(x) =: [x = \emptyset \rightarrow \emptyset ; Pv(x) \rightarrow x ; \sim \text{even}(x) \rightarrow \emptyset ;$$

$$\oplus \rightarrow \text{max}(\text{maxpv}(a(x)), \text{maxpv}(d(x)))]$$

$$\text{nextpv}(x) =: [Pv(x) \rightarrow x ; \oplus \rightarrow \text{nextpv}(S(x))]$$

$$\text{newpv}(x) = \text{nextpv}(S(\text{maxpv}(x)))$$

As we introduced the new symbol * by adding the axiom $x * y = x \oplus y$, we now introduce eight new symbols by adding the following eight axioms.

$$Pv(x) \equiv Pv(x)$$

$$Iv(x) \equiv Iv(x)$$

$$Pfvb(x) \equiv Pfvb(x)$$

$$Ifvb(x) \equiv Ifvb(x)$$

$$\text{newpv}(x) = \text{newpv}(x)$$

$$\text{newiv}(x) = \text{newiv}(x)$$

$$\text{newpfvb}(x) = \text{newpfvb}(x)$$

$$\text{newifvb}(x) = \text{newifvb}(x)$$

We shall call these symbols the eight special function symbols, and these axioms the eight special function axioms.

Now that we have the symbols Pv, Iv, Pfvb, and Ifvb available, we can use them for an additional task. We can use them for the subscripts on

predicate and individual function variables in place of the symbols tv, I, Pf, and If which we have been using. These last four symbols can then be eliminated from the alphabet. Thus a symbol like Iv has two uses. It is sometimes used as a predicate name and sometimes as a subscript in variables. Confusion will never arise between the two uses since the use is always clear from context. As before, we will often abbreviate by omitting subscripts in variables when the argument-types are clear from context.

2.1.1.9 Elimination of S and \emptyset and Addition of qu.

We can now proceed with elimination of S. S occurs both as a component of numerals of the form \bar{n} and as a symbol not part of a numeral. We shall first introduce an alternative notation for numerals which allows us to write numerals without employing an S. Then we shall find a function expression (which we shall abbreviate as \boxed{S}) which does not contain S and which we can use as a replacement for S in the same way that we earlier used \circ as a replacement for + and \cdot as a replacement for \cdot . The new way of expressing numerals will depend on the fact that each numeral is the Gödel number of some expression. We will express the numeral \bar{n} by writing (qu, α) , where α is the S-expression whose Gödel number is n. qu is a new symbol of the language. In other words, we will allow ourselves to write $\tilde{\alpha}$ as (qu, α) . We thus avoid the use of S unless α contains an S. (qu, α) may be thought of as an individual constant which names α . In this sense, the qu fulfills the same function that quotation marks fulfill in some logical systems and that the symbol quote fulfills in LISP. When we finish this section, the form (qu, α) will be the only acceptable form for numerals to take in the expressions of our language.

Let us begin by insisting that \emptyset (i.e. $\bar{0}$) be written in that form wherever it occurs. Since zero is the Gödel number of nil, this means we now replace \emptyset with (qu, nil) wherever \emptyset occurs in expressions. Thus \emptyset may be

eliminated from our alphabet. In our discussion, however, we shall continue to write \emptyset as an abbreviation for (qu, nil) , though \emptyset is no longer a symbol of our alphabet. For the (qu, α) notation to work for all numerals as well as the old notation, we want a theorem of form $\tilde{\alpha} = (qu, \alpha)$ for each S-expression α . Now in our system, formulae of the following types are theorems, where α and β are S-expressions: $\tilde{\alpha} * \tilde{\beta} = \widetilde{(\alpha.\beta)}$;

$newpv(\tilde{\alpha}) = \tilde{\beta}$ (where β is the first propositional variable of index $>$ those in α);

$newiv(\tilde{\alpha}) = \tilde{\beta}$ (where β is the first individual variable of index $>$ those in α);

$newpfb(\tilde{\alpha}) = \tilde{\beta}$ (where β is the first predicate variable base of index $>$ those in α);

$newifvb(\tilde{\alpha}) = \tilde{\beta}$ (where β is the first individual function variable base of index $>$ those in α).

If $\tilde{\alpha} = (qu, \alpha)$ is to be provable for all S-expressions α , then the counterparts of the above theorems using the (qu, α) notation should also be provable. These counterparts are:

A. $(qu, \alpha) * (qu, \beta) = (qu, (\alpha.\beta))$;

B. $newpv((qu, \alpha)) = (qu, \beta)$ (where β is the first propositional variable of index $>$ those in α);

C. $newiv((qu, \alpha)) = (qu, \beta)$ (where β is the first individual variable of index $>$ those in α);

D. $newpfb((qu, \alpha)) = (qu, \beta)$ (where β is the first predicate variable base of index $>$ those in α); and

E. $newifvb((qu, \alpha)) = (qu, \beta)$ (where β is the first individual function variable base of index $>$ those in α).

Now let us not add the formulae of form $\tilde{\alpha} = (qu, \alpha)$ to the theorem set yet. (These formulae contain S and we are trying to eliminate S.) Let us first add the formulae of types A through E to the theorem set (they don't contain S, except perhaps inside the α or β) by adding to our rules of inference five rules which generate theorems of these forms. We call these five rules, rules A through E. (The reason we need these now is indicated below.)

Footnote: Using the notation for rules of inference described in Section 2.1.4.2, and used in Table 5, the rules would be

- A: $T(((qu, x) * (qu, y)) = (qu, (x * y)))$,
 B: $newpv(x) = y \supset T(newpv((qu, x)) = (qu, y))$,
 C: $newiv(x) = y \supset T(newiv((qu, x)) = (qu, y))$,
 D: $newpfvb(x) = y \supset T(newpfvb((qu, x)) = (qu, y))$, and
 E: $newifvb(x) = y \supset T(newifvb((qu, x)) = (qu, y))$.

We are now in a position to define the function expression which we shall use to replace S. This expression will be abbreviated as \boxed{S} . The symbol S does not appear anywhere in it, though the numerals \emptyset , (qu, p) , $(qu, (p))$, and $(qu, (p,p))$ do appear in it. We add \boxed{S} to the initial set of algorithmic names.

Footnote: Using the notation explained in Section 2.1.2, one way of writing such an \boxed{S} would be as follows: Let us abbreviate an S-expression of the form $(p. (p. (p. \dots (p. nil) \dots)))$, which contains n p's, as \hat{n} .

Note, $\hat{0}$ is nil. Now let us define a new function with an algorithmic function name which we shall abbreviate as gn . This function shall have the property that for any S-expression α , if n is the Gödel number of α then $gn(\alpha) = \hat{n}$ holds. (We write α instead of (qu, α) , following the notation developed later. Note: $\hat{0}$, \hat{nil} , \emptyset , and $\bar{0}$ are abbreviations of the same expression, namely (qu, nil))

gn could be defined as follows:

$$x \hat{=} y =: [x = \emptyset \rightarrow y; \textcircled{\text{p}} \rightarrow d(x) \hat{=} (\textcircled{\text{p}} * y)]$$

$$x \hat{\wedge} y =: [x = \emptyset \rightarrow \emptyset; \textcircled{\text{p}} \rightarrow y \hat{=} (d(x) \hat{\wedge} y)]$$

$$\widehat{exp}(x, y) =: [y = \emptyset \rightarrow \textcircled{\text{1}}; \textcircled{\text{p}} \rightarrow x \hat{\wedge} exp(x, d(y))]$$

For propositional variables we define

$$convert(y) =: [y = \emptyset \rightarrow \emptyset; \textcircled{\text{p}} \rightarrow \textcircled{\text{p}} * convert(d(y))]$$

$$pvindexx(x, y) =: [\sim Pv(x) \rightarrow \emptyset; a(y) = x \rightarrow convert(y);$$

$$\textcircled{\text{p}} \rightarrow pvindexx(x, newpv(y) * y)]$$

$$pvindex(x) =: pvindexx(x, \emptyset)$$

We similarly define: $ivindex$ for individual variables,

$pfvindex$ for predicate function variable bases,

and $Ifvindex$ for individual function variable bases.

From this it is easy to define

$index$ such that if an atom α has index m then $index(\textcircled{\alpha}) = \textcircled{m}$ is provable.

$$gn(x) =: [x = \emptyset \rightarrow \emptyset; atom(x) \rightarrow d(\textcircled{2}) \hat{\wedge} index(x)];$$

$$\textcircled{\text{p}} \rightarrow ((\textcircled{2}) \hat{\wedge} gn(a(x))) \hat{=} \textcircled{\text{1}} \hat{\wedge} \widehat{exp}(\textcircled{2}, gn(d(x)) \hat{=} \textcircled{\text{1}})]$$

Similarly the inverse function gn^{-1} could be constructed as follows:

$$pvconvert(x) =: [x = \emptyset \rightarrow \emptyset; \textcircled{\text{p}} \rightarrow newpv(pvconvert(d(x))) * pvconvert(d(x))]$$

$$pvindex^{-1}(x) =: a(pvconvert(x))$$

similarly for $ivindex^{-1}$, $pfvindex^{-1}$, $Ifvindex^{-1}$.

Then we can easily define $index^{-1}$

$$half(x) =: [x = \emptyset \vee x = \textcircled{\text{1}} \rightarrow \emptyset; \textcircled{\text{p}} * half(dd(x))]$$

$$even(x) =: x = \textcircled{\text{2}} \hat{\wedge} half(x)$$

$$oddfactor(x) =: [\sim even(x) \rightarrow x; \textcircled{\text{p}} \rightarrow oddfactor(half(x))]$$

$$twosexpp(x, y) =: [\sim even(x) \rightarrow y; \textcircled{\text{p}} \rightarrow twosexpp(half(x), p * y)]$$

$$twosexp(x) =: twosexpp(x, \emptyset)$$

$$\begin{aligned}
 gn^{-1}(x) = & [x = \emptyset \rightarrow \emptyset; \\
 & \sim even(x) \rightarrow index^{-1}(half(\langle p \rangle * x)); \\
 & \oplus \rightarrow gn^{-1}(half(d(oddfactor(x)))) * gn^{-1}(d(twoexp(x)))]
 \end{aligned}$$

Then we can define \boxed{S} by $\boxed{S}(x) =: gn^{-1}(\langle p \rangle * gn(x))$.

Now by virtue of the new rules that we have added (rules A - E above) if α and β are S-expressions then: $S(\tilde{\alpha}) = \tilde{\beta}$ holds, if and only if, $\boxed{S}((qu, \alpha)) = (qu, \beta)$ is a theorem. In such a case, $\boxed{S}((qu, \alpha)) = (qu, \beta)$ is proved by successive transformations of the right side of the theorem $\boxed{S}((qu, \alpha)) = \boxed{S}((qu, \alpha))$ according to our rules of reference. This process of transforming the right side into a constant, (i.e. into the form (qu, β)) is called "evaluating" the right side. The evaluation could not proceed to completion without the presence in the system of the theorems generated by the five rules of inference A - E that we just added to the system. The only reason we added those five rules when we did was specifically to permit the evaluation of $\boxed{S}((qu, \alpha))$.

So \boxed{S} behaves like a successor function name when it is applied to constants. To see that \boxed{S} is really a successor function consider the following: Definition: Let $\check{\alpha}$ stand for the S-expression α with \boxed{S} uniformly replaced by S and then each sub - S-expression of form (qu, β) uniformly replaced by $\tilde{\beta}$. Now if α is a theorem then $\check{\alpha}$ is a theorem, and can be proved in a proof in which qu doesn't appear except as (qu, nil) . That this is so can be seen by replacing \boxed{S} with S and (qu, β) with $\tilde{\beta}$ in the proof of α . The result can be easily converted into a proof of $\check{\alpha}$ by intercalating lines here and there. (The only interesting intercalation required is in a step of the proof of α in which something of form $\boxed{S}(\gamma)$ was expanded. But if $\boxed{S}(\gamma) = \alpha$ is a theorem (α being an expansion of $\boxed{S}(\gamma)$), then $S(\check{\gamma}) = \check{\alpha}$ is going to be a true statement in arithmetic and will be provable without use of qu. We can then use this theorem in the required intercalation.)

Since \boxed{S} names a successor function we see that \boxed{S} induces a linear ordering on the set of S-expressions, an ordering that reflects the particular Gödel numbering we decided to use. Had we used a different Gödel numbering, we would have had a different \boxed{S} . For example, suppose we pick a different Gödel numbering which is just like our old Gödel numbering except it is based on an indexing of the atoms which, while being otherwise the same indexing we had before, has the indices for the atoms \supset and \forall switched. Suppose we defined an \textcircled{S} based on this Gödel numbering. This also would behave like a successor function. Now the basic successor function of our system is S. This is the function about which we are proving theorems. Now how does it behave when applied to constants? This we have not specified. We have specified some things about its behavior on constants when we added rules A - E. But these things are true of both \boxed{S} and \textcircled{S} above. Let us add enough axioms so that $\boxed{S}(x) = S(x)$, is provable. This will end the ambiguity. Let's do this in a perhaps inelegant but certainly straightforward manner, by adding $\boxed{S}(x) = S(x)$ to the set of axioms.

Now $\tilde{\alpha} = (qu, \alpha)$ becomes a theorem for each S-expression α . Also, it is still true that if α is a theorem then so is $\check{\alpha}$ where the proof of $\check{\alpha}$ does not employ a qu , except in (qu, nil) , and also does not use the axiom $\boxed{S}(x) = S(x)$. Thus we have added no new theorems except those employing a (qu, β) where β is not nil. Hence our system must still be consistent.

Now since $\boxed{S}(x) = S(x)$ is a theorem, we can replace S with \boxed{S} in all our axioms without changing the theorem set at all. Now let's remove $\boxed{S}(x) = S(x)$ from the axiom set. (Deletion of an axiom can never ruin consistency.) Now no axioms contain S. S has become a useless symbol so we eliminate it from the alphabet.

We now have an explicitly self-describing system. We shall no longer regard (qu, α) as a numeral, but rather an individual constant which names the S-expression α . We think of the individual variables as ranging over S-expressions, rather than over numbers. Domains and ranges of functions are no longer ever sets of numbers, but are instead sets of S-expressions. Finally, the expressions of our language may all be thought of as themselves S-expressions.

2.1.1.10 Our Axiom Set. This completes our overview of the system and its derivation from formal arithmetic by additions to and subtractions from the alphabet. In the following sections we shall give a more precise description of the system. We shall specifically state the axioms and rules of inference of the system. It will be clear then how the intuitive meanings of the symbols introduced in this section are implemented.

In addition to making precise our description of the axiom system, we shall simplify some axioms and rules of inference somewhat so that the properties in which we are interested will be more evident. (This simplification, while convenient, is not necessary. We could easily implement the suggestions of this section directly. How this would be done will be obvious after looking at our simplified axiom system. Our simplified axioms and rules are listed in Tables 4 & 5. (The format of the tables will be discussed more fully later.)) For example, the Peano axioms using \boxed{S} have been simplified to their analogues using $*$. Along with this we have suppressed the ordering of the S-expressions via the Gödel numbering. This ordering is useless to us and we only introduced it in order to make the formal connection between our (qu, α) notation and formal arithmetic. The suppression of the ordering has allowed us to dispense with several old axioms and rules and replace them with new simpler ones. These new axioms and rules

were theorems and meta theorems in the old system, so the replacement creates no inconsistencies.

Footnote: The changes involved in the above simplification of the axioms and rules may be summarized as follows:

Axioms deleted:

Peano axioms for \boxed{S}

$$x * y = x \oplus y$$

eight special function axioms, new variables are variables
Section 2.1.1.8

Axioms added:

Peano axiom analogues for * 8 - 10 Table 4

disjointness of atom classes 15 Table 4

new variables are new 16 Table 4

new variables are new 17 Table 4

Rules deleted:

rules B - E Section 2.1.1.10

Rules added:

variables are variables 14 Table 5

different atoms are \neq 15 Table 5

The axioms & rules on the right are theorems when the set of axioms and rules includes those on the left. The axioms added can be proved from the axioms deleted. Roughly speaking, rule 14, Table 5 can be proved as a meta theorem, from rules A-E, Section 2.1.1.10, the eight special function axioms, and the Peano axioms for \boxed{S} ; rule 15, Table 5 can be proved, as a meta theorem, from rules A-E Section 2.1.1.10 and the Peano axioms for \boxed{S} .

Actually, in addition to making the simplification discussed above, we shall also introduce a complication. We add two new rules of inference (we will call them rules 18 and 19) and we also add a procedure for continually adding more rules of inference. This procedure will take advantage of the system's self-describing capability. The procedure will provide a means to implement the suggestions made in the introduction section with regard to heuristic generation. Since this addition is not a simplification we will have to show that it does not introduce any inconsistencies. In future sections we

will show this and will also demonstrate certain incompleteness properties which follow from our consistency preserving techniques.

2.1.2 Expressions and Their Abbreviation

2.1.2.1 Motivation for Abbreviation. As we mentioned, we find it convenient to define a set of so-called S-expressions and to write our syntax rules so that the expressions of our language are all S-expressions. This practice allows us to state our meta theorems and rules of inference more simply. In our discussion, however, we will not want to write out expressions of our language as S-expressions. We will want to abbreviate our expressions into a notation similar to the usual notation for an applied first order predicate calculus system like ours. (We have used this more usual notation in the preceding discussion.) Now of course we won't abbreviate all S-expressions. We do abbreviate those which are expressions of our language. We may abbreviate more or less as suits the purposes of our discussion. (Eg., $p \vee (\sim p)$ is an expression which is already in abbreviated notation, but it may be further abbreviated to $p \vee \sim p$ if we wish.)

The reader who is familiar with the usual first order predicate calculus notation is already familiar with most of our abbreviated notation. The reader has already been introduced to our use of the Church λ . Our use of label, cond (alternatively pcond), * (LISP cons), *a* (LISP car), and *d* (LISP cdr), is the same as their use in LISP, and our abbreviation of cond is just like that in LISP m-notation. (A brief example of the use of cond and label was given in Section 2.1.1.5)

We will want to be very explicit in stating our syntax rules so we will state them for unabbreviated expressions. Since we will be writing the expressions in abbreviated form, we need to first specify a procedure by which the reader can unambiguously arrive at the unabbreviated expression

if he is given the abbreviated form. Before we do this we shall give an explicit definition of the set of expressions of our language.

2.1.2.2 Definitions of Classes of Expressions of our Language.

The symbols of our alphabet are called atoms; they are given in Table 1. The structures we will be concerned with are S-expressions (not to be confused with expressions or wexpressions which we shall define later). As in LISP, ξ is an S-expression iff it is an atom or of the form $(\alpha.\beta)$ where α and β are S-expressions. As in LISP, we may abbreviate $(\alpha.\text{nil})$ as (α) wherever it occurs in an S-expression, and $(\alpha.(\beta_1,\beta_2,\dots,\beta_n))$ as $(\alpha,\beta_1,\beta_2,\dots,\beta_n)$ wherever it occurs in an S-expression. An S-expression not abbreviated this way is said to be written in dot notation. An S-expression abbreviated in this way is said to be in comma notation. (We shall make a practice of using comma notation.) An S-expression which can be abbreviated in this way into a form without dots is called a list.

I shall use lower case Greek letters to indicate strings of symbols with properly paired parentheses and brackets. (This is the way I used them above.)

A type is an S-expression of one of the following forms :

1. Pv
2. Iv
3. $(Pfvb, \alpha_1, \alpha_2, \dots, \alpha_n)$ where each α_i is a type
4. $(Ifvb, \alpha_1, \alpha_2, \dots, \alpha_n)$ where each α_i is a type

We define a predicate, whose name we abbreviate as *typep* true on any S-expression of one of the above four forms. We now define the subset of S-expressions which follow the syntactic rules of our language. These we shall call expressions. With each expression we will associate an S-expression of one of the above four forms. This S-expression will be called the expression's type.

An expression of type Pv is called a formula-type expression. An expression of type Iv is called a term-type expression. An expression whose type is of form $(Pfvb, \alpha_1, \alpha_2, \dots, \alpha_n)$ is called a predicate function -type expression. An expression whose type is of form $(Ifvb, \alpha_1, \alpha_2, \dots, \alpha_n)$ is called an individual function-type expression. Both predicate function-type expressions and individual function-type expressions are called function-type expressions.

The class of expressions is that class of S-expressions in which we will be interested. It is the class of "expressions of our language."

We first define a subclass of the class of expressions, called the class of variables. There are four sorts of variables.

Sorts of Variables	Type of the Variable
1. Propositional variables. These are atoms. (see Table 1)	Pv
2. Individual variables. These are atoms. (see Table 1)	Iv
3. Predicate variables; these are of form $(\pi, \alpha_1, \alpha_2, \dots, \alpha_n)$ where π is a predicate variable base (see Table 1) and each α_i is a type	$(Pfvb, \alpha_1, \alpha_2, \dots, \alpha_n)$
4. Individual function variables; these are of form $(\phi, \alpha_1, \alpha_2, \dots, \alpha_n)$ where ϕ is an in- dividual function variable base (see Table 1) and each α_i is a type	$(Ifvb, \alpha_1, \alpha_2, \dots, \alpha_n)$

An expression is an S-expression of one of the following forms.

Form of the Expression	Type of the Expression
1. $\textcircled{\text{P}}$	Pv
2. $\textcircled{\text{I}}$	Pv
3. α ; (where α is a propositional variable)	Pv
4. α ; (where α is an individual variable)	Iv
5. (qu, α) ;(where α is an S-expression)	Iv
6. \supset	(Pfvb, Pv, Pv)
7. $=$	(Pfvb, Iv, Iv)
8. Pv	(Pfvb, Iv)
9. Iv	(Pfvb, Iv)
10. Ifvb	(Pfvb, Iv)
11. Pfvb	(Pfvb, Iv)
12. $(\pi, \alpha_1, \alpha_2, \dots, \alpha_n)$;(where π is a predicate function, variable base and each α_i is a type. This expression is a predicate variable.)	$(\text{Pfvb}, \alpha_1, \alpha_2, \dots, \alpha_n)$
13. *	(Ifvb, Iv, Iv)
14. newpv	(Ifvb, Iv)
15. newiv	(Ifvb, Iv)
16. newpfvb	(Ifvb, Iv)
17. newifvb	(Ifvb, Iv)
18. $(\phi, \alpha_1, \alpha_2, \dots, \alpha_n)$ (where ϕ is an individual function variable base and each α_i is a type. This expression is an individual function variable.)	$(\text{Ifvb}, \alpha_1, \alpha_2, \dots, \alpha_n)$

Form of the Expression	Type of the Expression
19. $(\pi, \alpha_1, \alpha_2, \dots, \alpha_n)$;(where π is an expression of type $(Pfvb, \beta_1, \beta_2, \dots, \beta_n)$ and each α_i is an expression of type β_i)	Pv
20. $(\theta, \alpha_1, \alpha_2, \dots, \alpha_n)$;(where θ is an expression of type $(Ifvb, \beta_1, \beta_2, \dots, \beta_n)$ and each α_i is an expression of type β_i)	Iv
21. $(pcond, (\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n))$;(where each α_i and β_j is an expression of type Pv)	Pv
22. $(cond, (\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_n, \beta_n))$;(where each α_i is an expression of type Pv and each β_j is an expression of type Iv)	Iv
23. $(\forall, (\eta_1, \eta_2, \dots, \eta_n), \alpha)$;(where each η_n is an individual variable and α is an expression of type Pv)	Pv
24. $(\exists, (\eta_1, \eta_2, \dots, \eta_n), \alpha)$; (where each η_i is an individual variable and α is an expression of type Pv)	Pv
25. $(\exists!, (\eta), \alpha)$; (where η is an individual variable and α is an expression of type Pv)	Pv

Form of the Expression	Type of the Expression
26. $(\iota, (\eta), \alpha)$; (where η is an individual variable and α is an expression of type Pv)	Iv
27. $(\lambda, (\eta_1, \eta_2, \dots, \eta_n), \alpha)$; (where each η_i is a <u>variable</u> of type β_i , and α is an expression of type Pv)	$(Pfvb, \beta_1, \beta_2, \dots, \beta_n)$
28. $(\lambda, (\eta_1, \eta_2, \dots, \eta_n) \alpha)$; (where each η_i is a <u>variable</u> of type β_i , and α is an expression of type Iv)	$(Ifvb, \beta_1, \beta_2, \dots, \beta_n)$
29. $(\text{label}, \pi, (\lambda, (\eta_1, \eta_2, \dots, \eta_n), \alpha))$; (where each η_i is a <u>variable</u> of type β_i , α is an expression of type Pv, and π is a <u>variable</u> of type $(Pfvb, \beta_1, \beta_2, \dots, \beta_n)$)	$(Pfvb, \beta_1, \beta_2, \dots, \beta_n)$
30. $(\text{label}, \phi, (\lambda, (\eta_1, \eta_2, \dots, \eta_n), \alpha))$; (where each η_i is a <u>variable</u> of type β_i , α is an expression of type Pv, and ϕ is a <u>variable</u> of type $(Ifvb, \beta_1, \beta_2, \dots, \beta_n)$)	$(Ifvb, \beta_1, \beta_2, \dots, \beta_n)$

2.1.2.3 Format for Abbreviation Rules. Although we are mainly interested in abbreviating expressions, there are a few situations when we may want to abbreviate an S-expression which, although composed of expressions, is not itself an expression. For this reason our procedures will be given for handling abbreviations of any S-expression .

We shall give a recursive procedure for deriving an S-expression from an abbreviated S-expression. This will be called the unabbreviating procedure. The procedure will be given recursively; i.e., we shall assume that we know how to unabbreviate any abbreviation shorter than the abbreviation we are working with. The procedure for unabbreviating single symbols will be given explicitly.

By the unabbreviating procedure we shall give in Sections 2.1.2.4, 2.1.2.5 and 2.1.2.6, an abbreviation is converted into an S-expression written in comma notation. (The comma notation was introduced at the beginning of Section 2.1.2.2 and used throughout 2.1.2.2 in defining the classes of expressions of our language.) In Section 2.1.2.7 we shall give the procedure for converting comma notation into dot notation.

2.1.2.4 Abbreviations of a Single Color With No "Defined" Symbols (The First 12 Rules). The symbols used in abbreviated S-expressions

are the symbols of the alphabet and

(v
)	^
,	~
.	∅
[a
]	d
;	
→	

(Certain symbols will appear as subscripts in an abbreviation) Other symbols may be used if already "defined" We shall ignore these for now and discuss them later. (A single word is, for our purposes, regarded as a single symbol; e.g., nil) The symbols may be in any number of colors. For now we shall limit ourselves to S-expression abbreviations written all in one color.

Abbreviated formula-type expressions look just like formulae in an applied first order predicate calculus (which, in fact, they are). As is usual in such formulations we may drop sets of parentheses, relying on precedence conventions to give us the information which the parentheses normally would. Our first task, then, is to replace these parentheses.

A substring of the string of symbols forming the abbreviation is called a sub-S-expression candidate if its parentheses match, if it can be unambiguously unabbreviated by our procedure, and if the result is an S-expression. A substring of the string of symbols forming the abbreviation is called a sub-expression candidate if its parentheses match, if it can be unambiguously unabbreviated by our procedure, and if the result is an expression. By our induction assumption, we see that we can determine all sub-S-expression candidates and subexpression candidates of our expression.

In the rest of Section 2.1.2, ξ denotes our abbreviation; $\sigma, \sigma_1, \sigma_2 \dots$ etc. denote sub-S-expression candidates. Greek letters other than ξ, λ , or σ denote subexpression candidates. A letter or string with a bar over it denotes the unabbreviated form of whatever is under the bar. () indicates optional parentheses.

We unabbreviate our expression ξ according to the following rules which we try to apply in order:

- 1A. If ξ is of form $(\sigma_1, \sigma_2, \dots, \sigma_n)$ ($n \geq 1$)
then $\bar{\xi}$ is $(\bar{\sigma}_1, \bar{\sigma}_2, \dots, \bar{\sigma}_n)$

- 1B. If ξ is of form (σ_1, σ_2) then $\bar{\xi}$ is of form $(\bar{\sigma}_1, \bar{\sigma}_2)$.
2. If ξ is of form $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ and $\bar{\phi}$ is a function-type expression, then $\bar{\xi}$ is $(\bar{\phi}, \bar{\alpha}_1, \bar{\alpha}_2, \dots, \bar{\alpha}_n)$.
3. If ξ is of form $[\alpha_1 \rightarrow \beta_1; \alpha_2 \rightarrow \beta_2; \dots; \alpha_n \rightarrow \beta_n]$ and $\bar{\beta}_1$ is a formula-type expression, then $\bar{\xi}$ is $(\text{pcond}, (\bar{\alpha}_1, \bar{\beta}_1), (\bar{\alpha}_2, \bar{\beta}_2), \dots, (\bar{\alpha}_n, \bar{\beta}_n))$.
4. If ξ is of form $[\alpha_1 \rightarrow \beta_1; \alpha_2 \rightarrow \beta_2; \dots; \alpha_n \rightarrow \beta_n]$ and $\bar{\beta}_1$ is a term-type expression, then $\bar{\xi}$ is $(\text{cond}, (\bar{\alpha}_1, \bar{\beta}_1), (\bar{\alpha}_2, \bar{\beta}_2), \dots, (\bar{\alpha}_n, \bar{\beta}_n))$
5. If ξ is of form $\{\alpha_1 \vee \alpha_2 \vee \dots \vee \alpha_n\}$ where each $\bar{\alpha}_i$ is a formula-type expression and no α_i has a \vee outside of parentheses, then $\bar{\xi}$ is $(\vee, \bar{\alpha}_1, (\vee, \bar{\alpha}_2, (\dots (\vee, \bar{\alpha}_{n-1}, \bar{\alpha}_n) \dots)))$.
6. If ξ is of form $\{\alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_{n-1} \wedge \alpha_n\}$ where each $\bar{\alpha}_i$ is a formula-type expression and no α_i has a \wedge outside of parentheses, then $\bar{\xi}$ is $(\wedge, \bar{\alpha}_1, (\wedge, \bar{\alpha}_2, (\dots (\wedge, \bar{\alpha}_{n-1}, \bar{\alpha}_n) \dots)))$.
7. If ξ is of form $\{\alpha_1 \phi \alpha_2\}$ where ϕ is an infix binary function name abbreviation (e.g., \supset , See definition in Section 2.1.2.5) and where the type of $\bar{\phi}$ is of the form $(\gamma, \beta_1, \beta_2)$ where β_1 is the type of $\bar{\alpha}_1$ and β_2 is the type of $\bar{\alpha}_2$. then $\bar{\xi}$ is $(\bar{\phi}, \bar{\alpha}_1, \bar{\alpha}_2)$.
8. If ξ is of form $\sim\alpha$ then $\bar{\xi}$ is $(\bar{\sim}, \bar{\alpha})$.

9. If ξ is of form $(\eta(\gamma_1, \gamma_2, \dots, \gamma_n) \delta)$ where η is a listbinder (see Table 1), then $\bar{\xi}$ is $(\eta, (\epsilon_1, \epsilon_2, \dots, \epsilon_n), \bar{\delta})$ where for each i if γ_i is a predicate variable base or an individual function variable base and if $\gamma_i(\alpha_1, \alpha_2, \dots, \alpha_m)$ or $(\gamma_i, \alpha_1, \alpha_2, \dots, \alpha_m)$ is a subexpression candidate of δ with β_j being the type of α_j , then ϵ_i is $(\gamma_i, \beta_1, \beta_2, \dots, \beta_m)$, and otherwise ϵ_i is γ_i .
10. If ξ is of form $\Pi(\alpha_1, \alpha_2, \dots, \alpha_n)$ where Π is a predicate variable base or individual function variable base, and β_i is the type of $\bar{\alpha}_i$, then $\bar{\xi}$ is $((\Pi, \beta_1, \beta_2, \dots, \beta_n), \bar{\alpha}_1, \bar{\alpha}_2, \dots, \bar{\alpha}_n)$.
11. If ξ is of form $\Pi_{\tau_1, \tau_2, \dots, \tau_n}$ where Π is either a predicate variable base, or is an individual function variable base, or is one of the four atoms, Pv, Iv, Pfvb, or Ifvb and where each $\bar{\tau}_i$ is a type, then $\bar{\xi}$ is $(\Pi, \bar{\tau}_1, \bar{\tau}_2, \dots, \bar{\tau}_n)$.
- 12A. If ξ is an atom
then $\bar{\xi}$ is ξ .
- 12B. If ξ is \emptyset , then $\bar{\xi}$ is (qu, nil) .
13. If ξ is a single symbol, not an atom,....

this situation will be discussed in section 2.1.2.5.

Note: $\forall(x)(P(x))$ is not an abbreviated expression; it unabbreviates to $(\forall, (x), (((P, Iv), x)))$. $\forall(x)P(x)$ is an abbreviated expression; it unabbreviates to $(\forall, (x), ((P, Iv), x))$.

Since rule 7 precedes 9,

$\forall(x) P(x) \supset P(x)$ unabbreviates to $(\supset , (\forall , (x) , ((P, Iv), x)) , ((P,Iv),x))$
as does $(\forall(x) P(x)) \supset P(x)$.

However

$\forall(x) (P(x) \supset P(x))$ is different; it unabbreviates to
 $(\forall , (x) , (\supset , ((P, Iv), x) , ((P, Iv) , x)))$.

2.1.2.5 "Defined" Symbols (Rule 13). A definition

statement is a symbol string in one of the following 5 forms:

- 1. $\phi =: \gamma$ where ϕ does not occur in γ ,
- 2. $\phi(\alpha_1, \alpha_2, \dots, \alpha_n) =: \beta$ where ϕ does not occur in β ,
- 3. $\alpha_1 \phi \alpha_2 =: \beta$ where ϕ does not occur in β ,
- 4. $\phi(\alpha_1, \alpha_2, \dots, \alpha_n) =: \beta$ where ϕ does occur in β , or
- 5. $\alpha_1 \phi \alpha_2 =: \beta$ where ϕ does occur in β .

In addition, the following must hold for the above 5 forms:

Each α_i is a variable - or variable base (predicate variable base or individual function variable base). Each ϕ is a single symbol (e.g. letter or word) which is not any of the symbols of our alphabet (Table 1) and not any of the symbols in the box below.

() , . [] ; →

The symbol ϕ occurring in a definition statement is called a "defined" symbol, and it is said to be defined by the definition statement in which it occurs. Any defined symbol may be used in an abbreviated S-expression (including the γ or β of a definition statement, see the 5 forms above), so long as it is defined by a definition statement occurring (or referred to) earlier in the discussion.

Rule 13 **now** can be written:

13A. If ξ is a single "defined" symbol ϕ then,

If ϕ is defined by a statement of form 1, $\bar{\xi}$ is $\bar{\gamma}$.

If ϕ is defined by a statement of form 2, $\bar{\xi}$ is $\overline{(\lambda, (\alpha_1, \alpha_2, \dots, \alpha_n), \beta)}$.

If ϕ is defined by a statement of form 3, $\bar{\xi}$ is $\overline{(\lambda, (\alpha_1, \alpha_2), \beta)}$.

If ϕ is defined by a statement of form 4, $\bar{\xi}$ is

$\overline{(\text{label}, (\theta, \delta_1, \delta_2, \dots, \delta_n), (\lambda, (\alpha_1, \alpha_2, \dots, \alpha_n), \gamma))}$

Where δ is the type of α_i and:

(1) If β is a formula-type expression then θ is the first predicate function variable base not occurring in β ;

(2) If β is a term-type expression then θ is the first individual function variable base not occurring in β ; and:

γ is β with each occurrence of ϕ replaced by $(\theta, \delta_1, \delta_2, \dots, \delta_n)$.

If ϕ is defined by a statement of form 5, $\bar{\xi}$ is as for form 4 with $n = 2$.

(With respect to rule 7:

ϕ is an infix binary function name abbreviation if and only if it is $\supset, =, *,$ or a "defined" symbol whose definition statement is of form 3 or 5 above.) The reader may note the striking similarity between our abbreviated definition statements and LISP definition statements in m-notation. Obviously, the form of abbreviations is quite dependent on preceding definition statements.

13B. If ξ is of form *addad* (α), then $\bar{\xi}$ is $(\bar{a}, (\bar{d}, (\bar{d}, (\bar{a}, (d, \alpha))))))$.

Similarly for any string composed of a 's and d 's .

13C. If ξ is of form *addad*, then $\bar{\xi}$ is $(\lambda, (x), (\bar{a}, (\bar{d}, (\bar{d}, (\bar{a}, (\bar{d}, x))))))$.

Similarly for any string composed of a 's and d 's.

Example: \sim is an abbreviation for $(\lambda, (p), (\supset, p, \textcircled{\alpha}))$. (The definition statement of \sim is given in Table 3.) $\sim p$ is an abbreviation for $((\lambda, (p), (\supset, p, \textcircled{\alpha})), p)$.

2.1.2.6 Abbreviations Using Colored Symbols

(rules 14 - 17). In abbreviations we may use colored symbols to indicate a quote operation (i.e., to indicate the existence of a qu).

Color of Symbol	Meaning
Black	Unquoted symbol
Blue	Quoted symbol
Red	Doubly quoted symbol
Green	Triply quoted symbol
.	.
.	.
.	.
.	.

We will use colored Greek letters (except λ) in our rules in the following way: If α stands for a

string of symbols, then $\textcircled{\alpha}$ stands for the same string of symbols but with the following color changes:

Symbols black in α are blue in $\textcircled{\alpha}$;

Symbols blue in α are red in $\textcircled{\alpha}$;

Symbols red in α are green in $\textcircled{\alpha}$; etc., etc. Similarly, with other

Greek letters used to represent strings. If $\bar{\alpha}$ is β then we will sometimes

write $\textcircled{\bar{\alpha}}$ for $\textcircled{\beta}$. E.g., if α is $\textcircled{x} * y$ then $\bar{\alpha}$ is $(*, (\text{qu}, x), y)$,

$\textcircled{\alpha}$ is $\textcircled{\textcircled{x}} * \textcircled{y}$, and $\textcircled{\bar{\alpha}}$ is $\textcircled{(*, (\text{qu}, \textcircled{x}), y)}$. Note, $\textcircled{\alpha}$ stands for a string

which has no black symbols. If $\textcircled{\alpha}$ stands for such a string, then $\textcircled{\textcircled{\alpha}}$ stands for the same string but with the following color change:

symbols blue in $\textcircled{\alpha}$ are red in $\textcircled{\textcircled{\alpha}}$;

symbols red in $\textcircled{\alpha}$ are green in $\textcircled{\textcircled{\alpha}}$; etc., etc.

These color conventions given for α will be the same for other Greek letters (except λ) including σ . To handle colored abbreviations, we add the following rules.

14. If ξ is of form σ then $\bar{\xi}$ is $(qu, \bar{\sigma})$

15. If ξ is of form $\langle \sigma \rangle$ then $\bar{\xi}$ is $(*, \bar{\sigma}, \bar{\emptyset})$

16. If ξ is of form $\langle \sigma_1 \sigma_2 \sigma_3 \dots \sigma_n \rangle$ then $\bar{\xi}$ is $(*, \sigma_1, \overline{\langle \sigma_2 \sigma_3 \dots \sigma_n \rangle})$

17. Suppose there exist two sequences: $\delta_1, \delta_2, \dots, \delta_n$, a sequence of term-type expressions, and $\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n$, a sequence of variables, such that no ε_i occurs in ξ and such that a uniform simultaneous substitution of ε_i 's for δ_i 's in ξ yields α where $\bar{\alpha}$ is an expression. Now let β be the result of uniformly simultaneously substituting the δ_i 's for the ε_i 's in $\bar{\alpha}$. Then $\bar{\xi}$ is $\bar{\beta}$.

Example 1. $\heartsuit x$ (There is a definition statement for \sim in Table 3)

Use rule 17 with $\delta_1 = x$ and $\varepsilon_1 = p$. Then β is $\langle \langle (\lambda, (p), (\supset, p, \emptyset)) \rangle, x \rangle$ and the answer is $\bar{\beta}$ which we get by rule 16:

$(*, \overline{\langle (\lambda, (p), (\supset, p, \emptyset)) \rangle}, \overline{\langle x \rangle})$

rule 14: $(*, (qu, \overline{\langle (\lambda, (p), (\supset, p, \emptyset)) \rangle}), \overline{\langle x \rangle})$

rule 15: $(*, (qu, \overline{\langle (\lambda, (p), (\supset, p, \emptyset)) \rangle}), (*, \bar{x}, \bar{\emptyset}))$

rule 1, several times: $(*, (qu, (\bar{\lambda}, (\bar{p}), (\bar{\supset}, \bar{p}, \bar{\emptyset})), (*, \bar{x}, \bar{\emptyset}))$

rule 12, several times: $(*, (qu, (\lambda, (p), (\supset, p, \emptyset)), (*, x, (qu, nil)))$

This is a term-type expression, and is of type Iv. Note $\overline{\heartsuit x}$ is the same expression as $\overline{\langle \sim, x \rangle}$ so we say $\heartsuit x$ is $\langle \sim, x \rangle$ since they are abbreviations of the same expression. We shall frequently talk in this way. We can also say $\heartsuit x$ is $(*, \heartsuit, (*, x, \emptyset))$.

Example 2.

$$\heartsuit x * y$$

By example 1 we see $\heartsuit x$ is a subexpression candidate of type Iv.

Hence rule 7 operates: $(*, \overline{\heartsuit x}, \bar{y})$.

Rule 12: $(*, \overline{\heartsuit x}, y)$ This gives the same as (see Example 1)

$$(*, (*, \heartsuit, (*, x, \emptyset)), y) .$$

Example 3.

$$\heartsuit (x * y)$$

Now rule 7 doesn't work, but rule 17 does, as in Example 1 with

$\delta = x * y$. The result is the same as (see Example 1) $(*, \heartsuit, (*, (*, x, y), \emptyset))$.

This is quite different from Example 2.

Note that: $\heartsuit ((*, x, y))$ gives the same result so we can also say,

$$\heartsuit (x * y) \text{ is } \heartsuit ((*, x, y)) .$$

Example 4.

$$x \supset y$$

By rule 17 with $\delta_1 = x, \delta_2 = y, \epsilon_1 = p, \epsilon_2 = q$: $\overline{\supset (x, y)}$

Rule 16: $(*, \overline{\supset}, \overline{\supset (x, y)})$. Rule 16: $(*, \overline{\supset}, (*, \bar{x}, \overline{\supset}))$

Rule 15: $(*, \overline{\supset}, (*, \bar{x}, (*, \bar{y}, \emptyset)))$

Rule 12: $(*, \overline{\supset}, (*, x, (*, y, \emptyset)))$

Rule 14 and 12: $(*, (qu, \supset), (*, x, (*, y, (qu, nil))))$

Note: $x \supset y$ is $\supset (x * (y * \emptyset))$. It is also $\heartsuit (x \supset y)$.

Example 5.

$T(T(\overline{\supset (p, p)}))$ (There is a definition statement for T in the tables.)

By rule 2 and 14 alternately, we get:

$$\overline{\overline{T(T(\overline{\supset (p, p)}))}}$$

$$\overline{\overline{T(qu, T(\overline{\supset (p, p)}))}}$$

$$\overline{\overline{T(qu, \overline{\overline{T(\overline{\supset (p, p)})}})}}$$

$$\overline{\overline{T(qu, \overline{\overline{T(qu, \overline{\supset (p, p)})}})}}$$

$$\overline{\overline{T(qu, \overline{\overline{T(qu, (\supset, p, p))}})}} \quad \text{We then use rule 13.}$$

We must stress that all 17 unabbreviating rules are to be used as a unit. A symbol such as $\bar{\alpha}$ appearing in any rule means the result of successively applying all 17 rules to α .

2.1.2.7 Comma vs Dot Notation. The unabbreviating procedure given in Sections 2.1.2.4, 2.1.2.5, and 2.1.2.6, gives a result written in the so-called comma notation. This is a perfectly good notation and it is the one we shall generally employ (in fact, it is the notation we employed in Section 2.1.2.2 when we gave definitions of the classes of expressions), but it is itself a sort of abbreviation. (It is the result of making the kind of abbreviation mentioned at the very beginning of Section 2.1.2.2.) In other words, after applying the unabbreviating procedure of Sections 2.1.2.4, 2.1.2.5, and 2.1.2.6 to an abbreviated S-expression, one still has an abbreviated S-expression.

In order to completely unabbreviate an S-expression one must follow the above unabbreviating procedure with a second unabbreviating procedure which is given below. The notations here are the same as in the previous procedure. There are three rules.

1. If ξ is an atom, then $\bar{\xi}$ is ξ .
2. If ξ is of form (σ) , then $\bar{\xi}$ is $(\bar{\sigma}. \text{nil})$.
3. If ξ is of form $(\sigma_1, \sigma_2, \dots, \sigma_n)$ then $\bar{\xi}$ is $(\bar{\sigma}_1, \overline{(\sigma_2, \dots, \sigma_n)})$.

Note: These 3 rules are distinct from the preceding 17 rules. $\bar{\sigma}$ occurring in these 3 rules means the result of applying only these 3 rules to σ .

$\bar{\sigma}$ occurring in the preceding 17 rules means the result of applying only the preceding 17 rules to σ .

2.1.2.8 Reading the Expressions in the Tables and the Text. In Section 2.1.2.5 we pointed out that the form of an abbreviated expression depends very much on what definition statements precede it. We will want to use many "defined" symbols in the following sections. We could either defer each definition statement until the "defined" symbol is needed, or we could list all the definition statements now. We shall adopt the latter course for most "defined" symbols. However, instead of actually

inserting the block of definition statements into the text at this point, we have placed the block of definition statements in the tables in Section 4. We refer the reader to these definition statements now and warn the reader that we shall in the future freely use the symbols "defined" by the definition statements in Section 4 (except for Section 4.11).

To aid the reader in deciphering the tables of Section 4, we shall make the following remarks about the organization of the tables: The definition statements are all contained in Tables 3, 6, 7, 8, and 9. Tables 3, 6, and 9, consist entirely of definition statements. Since the definition statements are gathered in the tables for easy reference, the reader need not learn any of the definitions until he feels it is necessary. In fact, we shall (in Section 2.1.4.1) give English statements of the meanings of the important "defined" symbols. The reader will find these English statements sufficient for most purposes and he may decide that careful scrutiny of the definition statements is unnecessary.

Table 1 lists our alphabet.

Table 2 lists those function expressions which name our basic recursive functions, each of which has its own special evaluation procedure (see the note in Section 2.1.4.2). (In any machine employing our system each such procedure would be stored as a separate subroutine.) The meanings of the function expressions are given using the notation of Section 2.1.4.1.

Table 3: **This** is a basic list of definition statements for symbols which are abbreviations for the names of recursive functions. As in all tables except 11, each definition statement employs only those "defined" symbols which have been defined in previous definition statements. Table 3 gives definitions for all the "defined" symbols used in Tables 4 and 5 except T and *Pfstep*. We shall see in Section 2.1.4.2 that each of the function names given in Table 3

is an especially nice kind of function expression called a complete recursing function expression.

Tables 4 and 5: There are no definition statements in these two tables. Each of these tables is a sequence of abbreviated expressions, using the "defined" symbols which were defined in Table 3 (and also using *Pfstep* and T, "defined" in Tables 6 and 7; see apology below). The expressions abbreviated in Table 4 are the axioms of our system. We shall be discussing these in Section 2.1.4.3 and later sections. We shall see in Section 2.1.4.2, that our rules of inference can also be written as expressions in our language. Table 5 gives the abbreviations of these expressions. We shall be discussing them in Section 2.1.4.4 and later sections.

Table 6: Like Table 3, this table consists entirely of definition statements for symbols which are abbreviations of those especially nice kind of function expressions (which we shall discuss in Section 2.1.4.2) called complete recursing function expressions. As "defined" in this table, each $Rule_i$ symbol has an obvious relationship to the i 'th rule of inference in Table 4. At the end of the table, the $Rule_i$'s are used in a definition statement which "defines" the symbol *Pfstep* from which the symbol *Proof* (which names the predicate true on proofs) is immediately "defined." From this, in Table 7, the symbol T (which names the predicate true on theorems) is "defined." We shall discuss the meanings of *Pfstep*, *Proof*, and T in Section 2.1.4.1 and later sections. The whole purpose of the definition statements in Table 6 is to make simpler the definition statements for the above three predicate expressions.

Apology: Since the definitions in Table 6 can most easily be thought of as being derived in a natural way from the rules of inference in Table 5, we have placed the Table 6 definitions after the rules of inference. The reader

will find this arrangement more convenient than the reverse, but strictly speaking, since the expression T is used in the rules of inference, the definition statement for T , and hence all the statements in Table 6, should precede the rules of inference (in which T is used), and should also precede the axioms (in which $Pfstep$ is used).

Table 7: With this table we leave the realm of those nice complete recursing functions that we shall be discussing in Section 2.1.4.2. The definition statements in the remainder of the tables give abbreviations for functions that are in general not complete recursing. The only definition statement in Table 7 defines T , the predicate expression which names the predicate true on theorems. (We shall be discussing T at end of Section 2.1.4.1) Following the definition statement for T are several abbreviated expressions which use T . It will turn out that these expressions are theorems of our system and that their intended interpretations are virtually the same as the intended interpretations of the expressions in Table 5. It is intended that the reader defer examination of these theorems, and the other theorems in the tables, until after we have discussed the axioms and rules of inference in Section 2.1.4.

Table 8: Here we have several definition statements and interspersed theorems (proofs not given) which will be useful as a formal counterpart to our discussion of well formedness in Section 2.1.3.

Table 9: This table consists entirely of definition statements which are needed for the definition of apl which is defined at the end of the table. This function expression abbreviation will be discussed in Section 2.1.4.2.

Table 10: In this table are given several sample proof outlines. It is intended that the reader defer examination of this table until after we have

discussed the axioms and rules of inference in Section 2.1.4.

Table 11: This table has nothing to do with our axiomatic system. It contains some LISP-like routines for implementing an adaptive theorem prover as suggested in Section 3.

2.1.3 Well Formedness. In most axiomatic systems there is a rule of inference which allows uniform substitution of a well-formed formula for a propositional variable. What will be our analogue of this rule? We cannot allow substitution of any formula-type expression for a propositional variable, because, according to our definition, a formula-type expression may contain those "contradictory" algorithmic-names that we promised to exclude from theorems. For this reason we define a subset of the set of formula-type expressions, called the set of well-formed formulae. This set consists of just those formula-type expressions which contain no algorithmic-names except those which are generatable according to the procedure we discussed earlier. Thus the well-formed formulae will contain no "contradictory" algorithmic names. Because of the nature of our procedure for generating algorithmic-names, the set of well-formed formulae is not a decidable set. Consider the predicate expression abbreviated as F . (The symbol F is "defined" in Table 8) This expression names the predicate true on S-expressions which are well-formed formulae. Although, in form, F looks like an algorithmic name, the definition contains a function-type expression, (namely T) which can only be regarded as an ordinary-name since it contains a \mathcal{H} . Thus, F is not a pure algorithmic-name but a sort of hybrid, built up according to our rules of formation of predicate-type expressions from names, some of which are not algorithmic. Such hybrid names of individual functions and predicates are so common in our system that the distinction between ordinary-names and algorithmic-names, which we made earlier, is

actually of little use to us once we take the more complicated cases into consideration. We can still speak, however, of purely algorithmic names. Such names must reflect their algorithmic evaluation procedures. Thus their unabbreviated forms can contain no \forall , \exists , $\exists!$, or ι , except inside the expressions $\iota(y) ((atom(x) \wedge y = x) \vee (\exists(z) y * z = x))$ and $\iota(y) ((atom(x) \wedge y = x) \vee (\exists(z) z * y = x))$ which we are abbreviating as a and d respectively. (These two expressions are permitted only because any machine using our system would have stored a special algorithmic evaluation procedure for a and d — see note, Section 2.1.4.2.) F couldn't possibly be a purely algorithmic name since, being true on an undecidable set, it has no algorithmic evaluation procedure. Of course we can freely substitute well-formed formulae for propositional variables. That is: If α is a theorem, π is a propositional variable, and β is a well-formed formula, then the expression obtained by uniform substitution of β for π in α is also a theorem. This meta-theorem looks very much like the rule of inference we want to use. Close examination of the meta-theorem, however, shows that it cannot be a rule of inference, because there is no effective way of using it. How does one decide whether or not β is a well-formed formula? The set of well-formed formulae is not decidable. Yet we need some rule of inference similar to the above.

We define a subset of the set of well-formed formulae called the set of simple formulae. A simple formula is a well-formed formula whose unabbreviated expression contains no function-type expressions except atoms, predicate variables, or individual function variables. Note: all the "contradictory" algorithmic names (pure algorithmic names or not) are in the set of excluded function-type expressions.

The set of simple formulae is a decidable set, so the following can be a rule of inference: If α is a theorem, π is a propositional variable, and γ is a simple formula, then the expression obtained by uniform substitution of γ for π in α is also a theorem. If we have rules of inference which allow us to substitute already generated predicate function-type expressions for predicate variables and to substitute already generated individual function-type expressions for an individual function variable, we can, in fact, substitute any well-formed formula uniformly for a propositional variable by first substituting the appropriate simple formula and then replacing its predicate variables and individual function variables by the appropriate function-type expressions already generated. Later we will examine in detail the rules which allow us to do this.

We have taken a set of expressions (the set of formula-type expressions) and created two subsets: The first was created by excluding all expressions containing function-type expressions which were not generatable. (Non-generatable function-type expressions are all algorithmic names, but are not necessarily purely algorithmic.) The second was created by excluding all expressions containing function-type expressions which were neither atoms, predicate variables, nor individual function variables. We can similarly create two subsets for other particular sets of expressions, as the chart indicates.

<u>Particular Set of Expressions</u>	<u>First Subset</u>	<u>Second Subset</u>
Expressions	Well-formed expressions	Simple expressions
Form-type expressions	Forms	Simple forms
Formula-type expressions	Well-formed formulae	Simple formulae
Term-type expressions	Terms	Simple terms
Function-type expressions	Function expressions	Simple formations
Predicate function type expressions	Predicate expressions	Simple predicates
Individual function-type expressions	Individual function expressions	Simple individual functions

For each line of the chart the following facts hold:

The first subset is a subset of the set in Column one.

The second subset is a subset of the first subset.

The set in Column one and the second subset are both decidable.

The first subset is not decidable.

(Note that in our terminology a function expression is either an individual function expression or a predicate expression. Instead of "well-formed formula" we shall often write simply "formula.")

2.1.4 The Axiomatic System

2.1.4.1 Certain Functions and Relations. Having discussed the various kinds of expressions in which we shall be interested, we shall now proceed to discuss our axiomatic system in more detail. We shall specify the axioms and rules of inference of our system. In specifying these we shall make free use of the predicate and individual function symbols defined in the definition statements in the tables.

Before specifying our axioms and rules we will briefly discuss the defined predicate and individual function symbols that we shall be using in

abbreviations of the axioms and rules. The brief discussion will supplement the definitions in the tables by giving the intended interpretations of expressions whose abbreviations employ the defined individual function and predicate symbols. To do this economically we shall observe the following conventions: When a formula is true under the intended interpretation, we shall say that the formula holds. We shall say that an individual constant of form (qu, β) names the S-expression β . And when a formula of form $\alpha = (qu, \beta)$ holds, we shall say that α names the S-expression β even though α is not an individual constant. (For example, we say that $a((qu, (\alpha.\beta)))$ names α since $a((qu, (\alpha.\beta))) = (qu, \alpha)$ holds.) For any term-type expression α , we shall, in our discussion, write $\overline{\alpha}$ to mean the S-expression which α names. (e.g. We say that $\overline{a((qu, (\alpha.\beta)))}$ is α . Note that the symbol $\overline{\quad}$ is merely a convenience for purposes of discussion and is never part of an S-expression or S-expression abbreviation.)

In Section 2.1.3 we defined several useful classes of S-expressions. For each such class we can think of the unary predicate true on the class. Now the predicates of our language are defined over S-expressions so it is not surprising to find that we can write, in our language, algorithmic names for each of the above predicates. The tables give definition statements for defined symbols which are abbreviations for these algorithmic names.

We give below a chart which indicates intended interpretations of formulae whose abbreviations employ these defined symbols.

Formula	It Holds If and Only If α Names
<i>expression</i> (α)	an expression
<i>wfexpression</i> (α)	a well-formed expression
<i>simplexpr</i> (α)	a simple expression
<i>formtp</i> (α)	a form-type expression
<i>form</i> (α)	a form
<i>Ftp</i> (α)	a formula-type expression
<i>F</i> (α)	a well-formed formula
<i>simpleformula</i> (α)	a simple formula
<i>Tmtp</i> (α)	a term-type expression
<i>Tm</i> (α)	a term
<i>simpleterm</i> (α)	a simple term
<i>functiontp</i> (α)	a function-type expression
<i>function</i> (α)	a function expression
<i>Pfetp</i> (α)	a predicate function-type expression
<i>Pfe</i> (α)	a predicate expression
<i>Ifetp</i> (α)	an individual function-type expression
<i>Ife</i> (α)	an individual function name

Similarly, the several charts below give interpretations of formulae and terms whose abbreviations employ other defined symbols. (Note that an exact definition may be found of, for example, the set of simple terms, by first looking at the above chart and noting that the predicate true on simple terms is *simpleterm*, and then referring in Table 8 to the definition statement for *simpleterm* .) Also, two charts in Table 2 give interpretations

of formulae and terms which employ the basic complete recursive function expressions, which are listed in Table 2.

Use of Some Logical Connectives (Predication of Type (Pfvb, Pv) or (Pfvb, Pv, Pv)):

Formula	It Holds If and Only If
$\sim \alpha$	α does not hold.
$\alpha \wedge \beta$	α holds and β holds.
$\alpha \vee \beta$	α holds or β holds.
$\alpha \equiv \beta$	α and β both hold or else neither holds.

Use of Some Other Predicates:

Formula	It Holds If and Only If
$\alpha \neq \beta$	α and β name different S-expressions.
$\alpha \triangleleft \beta$	α names a sub-S-expression of the S-expression named by β .
$\alpha \triangleleft \beta$	$\alpha \triangleleft \beta$ holds and if α names a variable then it occurs free in the S-expression named by β . (This only makes sense when β names an expression.)
$\alpha \overset{1}{\triangleleft} \beta$	$\bar{\alpha}$ is a variable and $\bar{\beta}$ is an expression and if there is no free occurrence of $\bar{\alpha}$ in $\bar{\beta}$ occurring in any subexpression of $\bar{\beta}$ of form (label, π , δ). (It also holds in some cases when $\bar{\alpha}$ is not a variable or $\bar{\beta}$ is not an expression)
$freecheck(\alpha, \beta, \gamma)$	$\bar{\alpha}$ is a variable which doesn't occur free in $\bar{\gamma}$ inside the scope of a binder which binds a variable free in $\bar{\beta}$. (But it also holds sometimes when $\bar{\alpha}$ is not a variable)

<i>variable</i> (α)	α names a variable.
<i>nonvaratom</i> (α)	α names an atom which is not a variable.
$\alpha \in \beta$	β names a list of form $(\alpha_1 \alpha_2, \dots, \alpha_n)$ and $\bar{\alpha}$ is α_i for some n and i such that $1 < i < n$. (It may also hold in some cases when α doesn't name a list.)

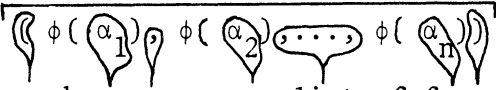
Use of Some Individual Functions:

Term	It Names
<i>expr:type</i> (α)	The type of $\bar{\alpha}$ whenever α names an expression.
<i>type</i> (α)	The type of $\bar{\alpha}$ if α names a variable, \emptyset if it doesn't.
<i>args</i> (α)	The list $(\beta_1, \beta_2, \dots, \beta_n)$ whenever α names a function-type expression of type $\Gamma_{\beta_1, \beta_2, \dots, \beta_n}$
<i>newvar</i> (α, β)	The variable γ where γ is of the same type as $\bar{\alpha}$ and does not occur in $\bar{\beta}$.

Use of Predicates Defined on Lists:

Formula	It Holds If and Only If
<i>andlista</i> (Π, α) (The type of Π must be $Pfvb_{IV}$ or this isn't a formula.)	$\Pi(\beta)$ holds for every β such that $\beta \in \alpha$ holds. (So α is meant to be a list)
<i>andlistlista</i> (Π, α, β) (The type of Π must be $Pfvb_{IV,IV}$ or this isn't a formula.)	α and β name lists of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$ and $(\beta_1, \beta_2, \dots, \beta_n)$ respectively, and $\Pi(\alpha_i, \beta_i)$ holds for each $i \leq n$. (It may also hold sometimes when α or β don't name lists as indicated.)

Use of an Individual Function Defined on a List:

Term	It Names
$maplistcar(\phi, \alpha)$ (The type of ϕ must be $Ifvb_{IV}$ or this isn't a formula)	 <p>when α names a list of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$.</p>

Use of Individual Functions Which Perform Substitutions:

Term	It Names the Expression Obtained by
$S(\alpha, \beta, \gamma)$	Uniform substitution of $\bar{\beta}$ for $\bar{\alpha}$ in $\bar{\gamma}$.
$Sf(\alpha, \beta, \gamma)$	Uniform substitution of $\bar{\beta}$ for all free occurrences of the variable $\bar{\alpha}$ in $\bar{\gamma}$.
$Snf(\alpha, \beta, \gamma)$	Uniform substitution of $\bar{\beta}$ for all occurrences of $\bar{\alpha}$ in $\bar{\gamma}$ except those occurrences inside sub-expressions which are function-type expressions.
$Ss\bar{l}(\alpha, \beta, \gamma)$	(Assuming α and β name lists of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$ and $\beta_1, \beta_2, \dots, \beta_n$ respectively) simultaneous uniform substitution of each β_i for α_i in $\bar{\gamma}$.
$Ssf\bar{l}(\alpha, \beta, \gamma)$	(Assuming α and β name lists of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$ and $(\beta_1, \beta_2, \dots, \beta_n)$ respectively and each α_i is a variable) simultaneous uniform substitution of each β_i for all free occurrences of α_i in $\bar{\gamma}$.

We have the three further predicates: *Pfstep*, *Proof*, and *T* .

Pfstep (α) holds if and only if

α names a list of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$ where α_1 is an axiom of our system or is derivable from the other α_i 's by means of our rules.

Proof (α) holds if and only if

α is a list of form $(\alpha_1, \alpha_2, \dots, \alpha_n)$ where $\alpha_n, \alpha_{n-1}, \dots, \alpha_2, \alpha_1$ is a sequence of expressions forming a proof in our system.

Now except for some in the first chart in this Section (2.1.4.1), all predicate and individual function names we have discussed so far in this section are purely algorithmic names ("purely algorithmic" is defined in Section 2.1.3.).

We define

$$T(x) =: \exists (y) (Proof(y) \wedge \alpha(y) = x) .$$

T does not have a purely algorithmic name. *T* (α) holds if and only if α names a theorem of our system.

2.1.4.2 The Meta Level. Consider a term $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$

where ϕ is an individual function expression without free variable and each α_i is either a function name, an individual constant (i.e. of form (qu, α)), or one of the two so-called propositional constants, \oplus and \otimes . Suppose this term names the S-expression β . Using LISP terminology, one could say that the term has value β or that the function expression ϕ gives value β when applied to the arguments $\alpha_1, \alpha_2, \dots, \alpha_n$. A LISP program [McCarthy 1962] is a function-type expression much like ϕ . Such a program is presented to the LISP interpreter which applies the program to the data, written in the form of arguments like $\alpha_1, \alpha_2, \dots, \alpha_n$ above.

The task of the interpreter is to evaluate terms

like $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ to produce the S-expression named by the term.

In our system we shall have a similar evaluation procedure. In a moment we shall direct the reader to the definition of our procedure. First, let us say that it differs from the LISP procedure in two ways:

A. In applying λ expressions, one substitutes the argument expressions directly into the matrix of the λ expression and the result is evaluated. (LISP would evaluate the arguments and substitute their values. Our scheme makes the LISP symbol FUNCTION, and all its complications, unnecessary.)

B. The result of evaluation is a constant. That is, it is ϕ or θ or a quoted expression, not the expression itself. Thus, in the above

example, the result of our evaluating $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ is β , where β names $\overline{\phi(\alpha_1, \alpha_2, \dots, \alpha_n)}$. We say $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ evaluates to β .

Now the evaluation procedure is a recursive procedure so if ϕ is the name of a non-recursive function, there will be choices of the arguments for which the evaluation procedure does not yield an individual constant which names $\overline{\phi(\alpha_1, \alpha_2, \dots, \alpha_n)}$. In such cases, either the procedure does not terminate, or it yields another term γ , which names $\overline{\phi(\alpha_1, \alpha_2, \dots, \alpha_n)}$, but which is not an individual constant. In the second case we still say $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ evaluates to γ . Our evaluation procedure is such that if ϕ is any purely algorithmic individual function name (defined in Section 1.3) then for any choice of arguments $\alpha_1, \alpha_2, \dots, \alpha_n$ (i.e. any choice of α_i 's such that: $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ is a term without free variables, each α_i which is a term is an individual constant, and each α_i which is a well-formed formula is a propositional constant), $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ will either evaluate to an individual constant or the evaluation procedure will not terminate. Such a $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ will never evaluate to a term which is not an individual constant. If $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ evaluates to an individual constant for any

such choice of arguments $\alpha_1, \alpha_2, \dots, \alpha_n$, then we say that ϕ is a complete recursing individual function expression. Any complete recursing individual function expression is a purely algorithmic individual function name.

Our evaluation procedure handles not only individual function expressions, but also predicate expressions, and analogous statements are true of them. If $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ is a well-formed formula, then either it evaluates to a formula γ which holds if and only if $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ holds, or else the evaluation procedure does not terminate. If ϕ is any purely algorithmic predicate name then for any choice of arguments $\alpha_1, \alpha_2, \dots, \alpha_n$ (ie. any choice of α_i 's such that: $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ is a well-formed formula without free variables, each α_i which is a term is an individual constant, and each α_i which is a well-formed formula is a propositional constant), $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ will either evaluate to a propositional constant or the evaluation procedure will not terminate. If $\phi(\alpha_1, \alpha_2, \dots, \alpha_n)$ evaluates to a propositional constant for any such choice of arguments $\alpha_1, \alpha_2, \dots, \alpha_n$, then we say that ϕ is a complete recursing predicate expression.

The complete recursing individual function expressions and the complete recursing predicate expressions are the so-called complete recursing function expressions.

Note: Each of the basic function expressions listed in Table 2 has a special recursive evaluation procedure. A machine employing our system would have a special little evaluation subroutine (analogous to LISP SUBR's) for each of the function names in Table 2. Thus, if ϕ is in Table 2, it is a complete recursing function expression. The function expressions given in Table 3 are built from those in Table 2 by operations analogous to composition and primitive recursion, so it is not hard to see that the function names given in Table 3 are complete recursing function expressions, as are the names given in Table 6. Of course, there are plenty of function expressions whose

abbreviations are given in Tables 7, 8, and 9 that are not complete recursing.

Our evaluation procedure will be written in a general way so that it will apply to any term or well-formed formula. If α is a form to which we apply our procedure, and if all function expressions in the form α are purely algorithmic names (Then they look like LISP programs.) and if α has no free variables, no quantifiers, and no λ , then either the evaluation procedure does not terminate or α evaluates to a constant (individual constant or propositional constant). If all function expressions in such a form α are complete recursing function expressions then α evaluates to a constant.

Consider the purely algorithmic individual function name which we shall abbreviate as apl . The definition statement of this defined symbol is given in Table 9. The intended interpretation is given below.

Term	It Names
$apl(\alpha)$	The expression which results from applying our evaluation procedure to α . (Note: this expression will not always be a constant.)

(apl is analogous to the LISP $eval$ function.) Thus, the reader will find in the tables the specification of our evaluation procedure. It is written there in the guise of a definition statement for apl . (So we describe our evaluation procedure in much the same way that the LISP 1.5 manual describes the LISP evaluation procedure.)

The following meta-theorems will hold in our system.

META THEOREMS: If α is a term which evaluates to β then $\alpha = \beta$
is a theorem of our system. If α is a well-formed formula which evaluates to β then $\alpha \equiv \beta$ is a theorem of our system.

Since our meta-theorems are statements about classes of S-expressions, we can write them as expressions in our language. For example, the first half of the above meta-theorem can be written

$(\text{Im}(x) \wedge \text{apl}(x) = y) \supset T(x \stackrel{\alpha}{=} y)$ where x is to be α and y is the β . Since rules of inference are meta-theorems, we can write them in the same way. Modus ponens is $(T(x \stackrel{\alpha}{=} y) \wedge T(x)) \supset T(y)$.

Now a rule of inference is a meta-theorem of form $\alpha \supset T(\beta)$ where α is a well-formed formula and β is a term. To apply the rule to a set of theorems $\{\eta_1, \eta_2, \dots, \eta_m\}$ we substitute constants for all free variables (These constants are the parameter values referred to in Section 1.5 in the introduction.), obtaining something of form $\epsilon \supset T(\delta)$. ϵ is then evaluated in

the normal way except that whenever something of the form $T(\gamma)$ is encountered it is replaced by \oplus if and only if the evaluation of γ terminates in something of form (qu, η_1) . If this procedure terminates and the result is \oplus , and if the evaluation of δ terminates in something of form (qu, μ) , then we say that a result of applying rule $\alpha \supset T(\beta)$ to antecedent set $\{\eta_1, \eta_2, \dots, \eta_m\}$ is the theorem μ . If any of the above conditions fail to be met, we say the attempt to apply the rule fails.

With this application procedure in mind, we have used the above notation to write, in Table 5, the complete list of the initial rules of inference for our system. (Remember, we have a procedure for adding new rules of inference. This is explained in Section 2.2.1.2) Note that a rule will be more useful the fewer function expressions it contains which are not complete recursing, since if it contains many function expressions which are not complete recursing function expressions, the evaluation procedures will tend to not terminate or to yield something other than a constant.

2.1.4.3 The Axioms: (Table 4). Furnished with the meanings of the defined functions and predicates: *atom*, *nonvaratom*, \sim , \vee , \wedge , \equiv , \leq , and *Pfstep*, as well as the meanings of the primitive symbols of the language, we are ready to examine our set of axioms listed in Table. 4. For purposes of our examination we divide the axioms into several groups.

Axioms 1 - 5: This is simply one formulation of the axioms for a finite axiomatization of the pure first order predicate calculus. These are just as they might be in a formal arithmetic.

Axioms 6 - 7: These are the axioms of equality. Again these are just as they might be in formal arithmetic. Axiom 7b is needed for the substitutivity of equivalence discussed in Section 2.2.1.1. The only reason this \equiv counterpart of 7a appears in our system, but not in most formal arithmetics, is that we have predicates with domain of truth values as well as predicates with domain of individuals.

Axioms 8 - 10: These are our analogues of the Peano axioms, where our axioms are based on $*$ rather than on \boxed{S} . Recall that we made this simplification with assurance that we were only decreasing our power by doing so. Had we retained the more powerful axioms based on \boxed{S} , they would have been:

$$\sim(\emptyset = \boxed{S}(x))$$

$$\boxed{S}(x) = \boxed{S}(y) \supset x = y$$

$$(P(\emptyset) \wedge \forall(x)(P(x) \supset P(\boxed{S}(x)))) \supset \forall(x)P(x)$$

Axioms 11 - 13: We could regard \odot , \exists , and $\exists!$ as abbreviations and not actual symbols of our language. However, we do actually introduce them as symbols, and these axioms may be regarded as their definitions.

Axiom 14: This is the definition of ι . ι cannot be regarded as an abbreviation. We need the actual symbol or we lose power to represent the functions we want represented. Note we have only formalized an interpretation of $\iota(x) P(x)$ when in fact $\exists!(x) P(x)$ holds. We could formalize an interpretation in the other cases by picking an arbitrary interpretation, say nil, and adding the axiom

$$\sim\exists!(x)P(x) \supset \iota(x)P(x) = \emptyset$$

Axioms 15, 16, and 17: In Section 2.1.1.10 we removed from our system the ordering of the atoms via indexing. We retained, however, a few axioms (previously theorems) and rules which gave the minimal amount of information about the atoms needed to carry out the proofs we want. These three are the axioms retained.

Axiom 18: Remember that we need certain algorithmic names in our initial set of algorithmic names. This set is the set of algorithmic names which appear in the axioms. Now all algorithmic names which we need in the initial set are in the unabbreviated expression which we abbreviate as *Pfstep*. Axiom 18 ensures that these algorithmic names are all in the initial set.

2.1.4.4 The Rules of Inference (Table 5). Furnished with the meanings of those defined functions and predicates listed in the charts of Section 2.1.4.1, as well as the meanings of the primitive symbols of the language, we are ready to examine the rules of inference listed in Table 5. These are written in the notation explained in Section 2.1.4.2. In Rules 2,6,16, and 17, the predicate λ is used to make sure that in no theorem is there, occurring inside a subexpression of form $(\text{label}, \pi, \delta)$, a variable bound by a binder located outside that subexpression. The λ in Rule 6 has another purpose too, which we discuss below. Note that Rules 3,4,5,16, and 17 are the only rules which can generate new expressions of form $(\text{label}, \pi, \delta)$. (I.e., they are the only rules that generate new algorithmic names.)

Rule 1: Modus ponens.

Rule 2: Generalization.

Rule 3: Change of bound variable.

Rule 4: Substitution of simple expression for a variable of the same type. Recall, we substitute simple expressions instead of well-formed

expressions because by using this procedure in conjunction with Rule 5, we lose no power, and have a rule we can effectively apply.

Rule 5: Substitution of function expression for function variable. This is written in a way that takes advantage of the following property of the function *exprtype* (defined in Table 3). (This property is not mentioned in the discussion of Section 2.1.4.1). If $T(\beta) \wedge \alpha \leq \beta$ holds, then α will actually be a function expression whenever $a(\text{exprtype}(\alpha))$ names either Pfvb or Ifvb. (The analogous statement for forms does not hold: i.e., simply having $T(\beta) \wedge \alpha \leq \beta$ hold and having $\text{exprtype}(\alpha)$ name Pv or Iv does not ensure that α is a form.) Thus, for Rule 5 to be successfully applied, the constant (i.e., parameter value) substituted for the free variable z in the rule must name a legitimate function expression.

Rule 6: Application of a λ function to its arguments. Axiomatizations of formal arithmetic which do not have a λ notation frequently have one rule which is, in effect, Rule 5 followed immediately by Rule 6 (so the λ disappears as soon as it is substituted in). Rule 6 and Rule 7 are consistent with our modifications of the LISP evaluation procedure. Examining in detail the statement of Rule 6, suppose η, υ & ω are the constants (i.e. parameter values) to be substituted for $y, u,$ and v respectively in the application of the rule. If η is not a legal term then neither $\eta \leq \upsilon$ nor $\eta \leq \omega$ can hold and thus, for the rule to be successfully applied, υ and ω must be identical expressions. Rule 6 allows us to simplify the expression $((\lambda, (x), x), y)$ to y inside a theorem. However, we don't want to simplify $\text{maplistcar}((\lambda, (x), x), y)$ (which we can also write as $(\text{maplistcar} . ((\lambda, (x), x), y))$) to $(\text{maplistcar} . y)$ inside a theorem. The result is not even an expression. To prevent this sort of misuse of functional arguments, the condition *expression*(v) is added to the statement of Rule 6. The condition

$(y \Leftarrow u \vee (andlista((\lambda(z) \sim z \Leftarrow adda(y)), ada(y)) \wedge$
 $andlistlista((\lambda(x,z) \textcircled{\text{label}} \Leftarrow z \supset x \Leftarrow adda(y)), ada(y), d(y))))$

in Rule 6 ensures that no new expressions of form $(\text{label}, \pi, \delta)$ are generated.

Rule 7: Function recursion. This shows the use of label in functions defined recursively. The $ad(y) \Leftarrow add(y)$ term in the rule assures us that no new label expressions can be generated by this rule.

Rules 8, 9, 10, and 11: Conditional expressions. When combined with the propositional calculus rules, Rule 11 says one can replace α with $[\oplus \rightarrow \alpha]$; Rule 9 says from $\sim \alpha \supset (\dots [\beta_1 \rightarrow \gamma_1; \beta_2 \rightarrow \gamma_2; \dots; \beta_n \rightarrow \gamma_n])$ and $\alpha \supset (\dots \delta \dots)$ we can infer $(\dots [\alpha \rightarrow \delta; \beta_1 \rightarrow \gamma_1; \beta_2 \rightarrow \gamma_2; \dots; \beta_n \rightarrow \gamma_n] \dots)$. The rules also let us go in the reverse direction. In LISP a conditional expression (beginning with a cond) can be, in effect, either a formula-type expression or a term type expression. In our system, we use the atom pcond in the formula-type expressions and reserve the atom cond for the term-type expressions.

Rule 12: Listbinder notation. This lets us write $\forall(\xi_1) \forall(\xi_2) \dots \forall(\xi_n) \alpha$ as $\forall(\xi_1, \xi_2, \dots, \xi_n) \alpha$. Similarly for \exists .

Rule 13: Definition of qu. See Section 2.1.1.9 for discussion.

Rules 14 and 15: Rules about atoms preserved along with atoms 15, 16, and 17 when, in Section 2.1.1.10, we got rid of the ordering on atoms.

Rules 16 and 17: These are the rules which generate new algorithmic names. They implement the procedures suggested in Section 2.1.1.5 for avoiding "contradictory" algorithmic names. We shall discuss these rules in detail in section 2.2.1.1. Note the following with regard

to the specific form of the rules : Suppose ζ and ω are the constants (i.e. parameter values) that we substitute for z and v respectively in applying these rules. Then Rules 16 and 17 require us to replace every occurrence of ζ in ω . This may seem like a restriction, but it is not, since

we can always trivially re-code the ζ 's we want to replace and then replace only re-coded ζ 's. This re-coding can be merely a change of a bound variable, accomplished via Rule 3. Note that any function expression inside the generated function expression must have previously been generated.

Rules 18 and 19: These rules are the central feature of our system. They allow us to shift theorems from the object level to the meta level and back again. We shall, in Section 2.2.1.2, discuss how these rules operate and why they don't destroy consistency. We shall constantly add new rules to the system according to a scheme which we shall discuss in Section 2.2.1.2.

2.1.4.5 Proofs and theorems. Table 10 consists of some sample proofs which utilize the various rules of inference. We shall be referring, from time to time, to lines in these sample proofs which illustrate interesting applications of the various rules.

Accompanying almost every line of proof in the Table is an English phrase indicating how the line was derived. This phrase usually gives the rule used and the previous lines to which the rule was applied. It may also state the constants (i.e. parameter values) which were substituted for the various free variables in the rule. These are given by writing strings of form $\xi =: \sigma$ where ξ is an individual variable occurring in the statement of the rule and σ is the individual constant which is to be substituted for it in this application of the rule. (Such strings are, of course, not definition statements, in spite of their appearance.)

For the sake of brevity, many lines have been omitted from the proofs in Table 10, so that Table 10 actually consists of proof outlines rather than complete proofs. In cases where many lines have been left out, an English phrase will indicate the nature of the omitted development. Sometimes we just indicate the key rule or key axiom used in the development. The phrase "by

P.C." indicates that the omitted development employs only those rules which our system shares with pure first order predicate calculus. The phrase is used rather loosely and really is a signal that the omitted development is trivial, uninteresting, and employs no techniques which are distinctive to our system.

In some of the later sample proof outlines major sections of the proof have been replaced by an English discussion of the principles involved. Use of Greek letters to stand for any one of a class of expressions makes these outlines into proof outline schemata, the proof outline being good no matter which particular expressions are substituted for the Greek letters.

It is not intended that the reader necessarily read through all of Table 10. We shall, however, frequently refer to parts of Table 10 which illustrate interesting points. The proof outlines in that table will be a useful source of examples in the coming discussion.

Since the lines of these proof outlines are theorems, we shall freely mention them as theorems in the text. We shall frequently mention other theorems in the text without giving an outline of the proof of each one. The methods one would use in constructing proofs of such theorems would not differ substantially from the methods already illustrated in Table 10. In Tables 7 and 8 we similarly mention some theorems without giving outlines of their proofs.

2.1.4.6 Summary. As promised in Section 2.1.1.1, we have given a formal description of all parts of our formal system. Our alphabet and formation conventions were described in Section 2.1.2. The axioms and rules of inference were given in Section 2.1.4. The well-formed expressions were given in Section 2.1.3. The intended interpretation was given in Section 2.1.1, especially Section 2.1.1.9.

In Section 2.1.1 we described many of the formal properties of our

system. In doing so we outlined the reason that our system was consistent (since it can be derived from formal arithmetic by consistency preserving transformations). Unfortunately the discussion in Section 2.1.1 preceded the description of our notation (given in Section 2.1.2) and our axioms and rules of inference (given in Section 2.1.4). Thus the discussion in Section 2.1.1 was necessarily incomplete, since we did not have the notation of the system available for use in the discussion.

Our remaining task is to discuss in detail some formal properties of our system which we were not able to discuss until now because we lacked the notation. The first such property is consistency.

2.2 Formal Arguments

2.2.1 Consistency

2.2.1.1 Generation of Function Expressions. As we said in the last section, most of the consistency argument was made in Section 2.1.1. We need only review the parts of that argument which were vague.

These are the two parts we will review:

(1) The procedure for generating new function expressions described in Section 2.1.1.5 for a modified formal arithmetic is followed essentially unchanged in our formal system. We shall discuss this in more detail in this Section (2.2.1.1), and show in detail why the procedure introduces no inconsistencies.

(2) In Section 2.1.1.10, we introduced Rules 18 and 19 which allow us to shift theorems from the object level to the meta level and back again. We also introduced a procedure for constantly adding more rules of inference. As yet we have done no more than mention this procedure and Rules 18 and 19. The procedure and the rules will be fully specified in Section 2.2.1.2 and the consistency argument will be completed.

Rules 18 and 19 together with the procedure for adding new rules are the crucial features needed for use in an adaptive theorem prover of the type we are discussing. Discussion of these crucial features is deferred until after the discussion, in Section 2.2.1.1, of the rest of the consistency argument.

That is, in Section 2.2.1.1 we shall be discussing the consistency not of our entire system but rather of a limited system which is without Rules 18 and 19 and without the procedure for adding rules, but which is otherwise just like our entire system. Once we have completed the consistency argument for the limited system, we shall show (in Section 2.2.1.2) how Rules 18 and 19 and our procedure for adding rules can be added to the limited system without destroying consistency.

In Section 2.2.1.1, then, we shall be discussing the procedure for generating new function expressions as it operates in the limited system. We can divide function-type expressions into those of form $(\lambda, (\xi_1, \xi_2, \dots, \xi_n), \beta)$, which we shall call λ expressions, and those of form $(\text{label}, \pi, \delta)$, which we shall call label expressions. λ expressions are easily generatable via Rule 6. It is the procedure for generating label expressions that chiefly concerns us here. This procedure was briefly discussed, for a modified formal arithmetic, in Section 2.1.1.5. (The conclusions given there about generatability still hold in our limited system.) The basis of the procedure is the use of Rules 16 and 17. Examples of the use of Rules 16 and 17 may be seen in Section 4.10.5 in several different lines.

What might cause consistency problems in the limited system? Generation of λ expressions causes no consistency problems. It is the generation of label expressions (i.e., function type expressions of form $(\text{label}, \pi, \delta)$) that we have to worry about. Note: Any λ expression in which no label expression occurs is easily generatable via Rule 6.

If no label expressions were ever generated we could transform all our theorems by application of all the λ expressions to their arguments. (E.g. by repeated application of Rule 6). This eliminates all λ expressions from the theorem. Such a transformation transforms a proof into a proof in a simpler system which uses no λ 's. In the simpler system the rule for substitution for a function variable is like the more familiar formulations such as the one in Church [Church 1956]. This simpler system is familiar to us and is consistent by the standard model theoretic proof. The above argument shows that our limited system is consistent if we promise never to generate a label expression.

Now, because of the extensive use of λ and *Snf* in our rules, label expressions can be generated in the limited system only by

- (A) use of axiom 17 or 18 in which label expressions occur.
- (B) use of Rules 16 or 17.
- (C) change of bound variable in a label expression via Rule 3.
- (D) substitution for a variable free in a label expression via Rules 4 or 5.

Consider a system which is identical to the limited system except that axioms 17 and 18 and Rules 16 and 17 have been eliminated. Let us call this the basic system. By the above argument, the basic system is consistent. It is easy to see that (but for the lack of axiom 17) the basic system is identical to the limited system except that no label expressions may be generated.

We shall prove the consistency of the limited system relative to the basic system. We postulate a sequence of systems $W1, W1', W2, W3, W3', W4', W4$ where $W4$ is the limited system and $W1$ is the basic system (with a minor addition). We shall show that each system in the sequence is consistent if the preceding system is, either by showing that a proof of \mathcal{Q} in the former can be converted into a proof of \mathcal{Q} in the latter, or by showing that the two systems have the same theorem set, or by showing that the former is identical to the latter but for an added axiom which cannot ruin consistency.

We shall now define the systems $W1, W1', W2, W3, W3', W4,$ and $W4'$ by stating how each one differs from the limited system: (1) which axioms and rules the limited system has that it has not, and (2) which axioms and rules it has that the limited system has not. (All these systems have the same alphabet.)

Axioms and rules in category (2) above are selected from the following list:

$$\text{Axiom } 14': \quad \sim \exists!(x) P(x) \supset \exists (x) P(x) = \emptyset$$

$$\begin{aligned} \text{Rule } 5': \quad & (T(y) \wedge T(u) \wedge z \leq u \wedge \text{variable}(x) \wedge \\ & \text{type}(x) = \text{exprtype}(z) \wedge \text{freecheck}(x, z, y)) \\ & \supset T(Sf(x, z, y)) \end{aligned}$$

$$\begin{aligned} \text{Rule } 4^u: \quad & (T(y) \wedge \text{variable}(x) \wedge \sim x \leq y \wedge \text{Simplexpr}(z) \wedge \\ & \text{type}(x) = \text{expertype}(z) \wedge \text{freecheck}(x, z, y)) \\ & \supset T(Sf(x, z, y)) \end{aligned}$$

$$\begin{aligned} \text{Rule } 5^u: \quad & T(y) \wedge T(u) \wedge z \leq u \wedge (Pfv(x) \vee Ifv(x)) \wedge \sim x \leq y \wedge \\ & \text{type}(x) = \text{exprtype}(z) \wedge \text{freecheck}(x, z, y)) \\ & \supset T(Sf(x, z, y)) \end{aligned}$$

$$\begin{aligned} \text{Rule } 5'^u: \quad & (T(y) \wedge T(u) \wedge z \leq u \wedge \text{variable}(x) \wedge \sim x \leq y \wedge \\ & \text{type}(x) = \text{exprtype}(z) \wedge \text{freecheck}(x, z, y)) \\ & \supset T(Sf(x, z, y)) \end{aligned}$$

Rule 20: (This one we shall state in English)

If β results from α by substitution of μ for ν at zero or more places (not necessarily all occurrences of μ in α), and if none of these substitutions occurred within a function expression, and if β is an expression, and if α is a theorem, and if either $\mu \equiv \nu$ or $\mu = \nu$ is a theorem, then β is a theorem.

System	Differs from limited system in			
	Deletion of		Addition of	
	Axioms	Rules	Axioms	Rules
W1	17,18	16,17	14'	none
W1'	17,18	16,17	14'	20
W2	none	16,17	14'	20
W3	none	4,5	14'	4 ^u , 5 ^u , 20
W3'	none	4,5	14'	4 ^u , 5' ^u , 20
W4	none	none	none	none
W4'	none	5	14'	5', 20

We shall now show the consistency of each of these systems by the methods we have mentioned.

Consistency of W1

Except for the addition of axiom 14', the system W1 is identical to the basic system, consistent by the standard model theoretic proof. The introduction of axiom 14' does not destroy this model theoretic proof, for, if $\exists!(x)\Pi(x)$ is interpreted to mean the unique x such that $\Pi(x)$ holds if such a unique x exists (This interpretation makes axiom 14 true and says nothing about axiom 14'.), and if $\exists!(x)\Pi(x)$ is interpreted to mean nil if such a unique x does not exist (This interpretation makes axiom 14' true and

says nothing about axiom 14.), then the interpretation is still consistent with the other axioms and the rules of inference.

Consistency of $W1'$

$W1'$ differs from $W1$ only in the addition of Rule 20. We can show that the theorem set of $W1'$ is identical to the theorem set of $W1$ by showing that Rule 20 may be proved as a meta-theorem for the System $W1$.

Rule 20 is our substitutativity of equivalence rule and it can be proved for $W1$ by the standard induction technique. For example, it can be proved by a method analogous to the method of Church's proof of his Corollary *342 [page 190, Church 1956]. We shall just mention a minor way in which our proof must differ from Church's. Church proves Corollary *342 from Theorem *340. Theorem *340 he proves by induction on the size of the formula. We do something similar. But because of the way $W1$ uses ι and cond, well-formed formulae may occur inside terms inside theorems. Hence, where Church uses \equiv in the statement of his Theorem *340, we must allow for either \equiv or $=$ in order for the induction to work. Axiom 14' is needed to carry the induction through for the case where Church's A begins with ι .

Consistency of $W2$

$W2$ differs from $W1$ only in the addition of Axioms 17 and 18. The effect of Axiom 18 is to introduce certain label expressions into the language, namely those which we wish to be in our initial set, and those identical to initial set members but for a change in bound variable.

Addition of any one of the label expressions in the initial set is very similar to addition of a new function constant. To see that the addition of the label expression causes no inconsistencies, let us consider the analogous problem in adding any new function constant symbol to the system.

One could modify the system W1 by adding function constants in the way we shall describe.

Suppose one wishes to add a symbol ϕ to the system and suppose one wishes it to be a function constant of type ρ , naming a certain function. The addition can be accomplished in two steps. First one adds ϕ to the alphabet and changes the expressions *Pfatom*, *Ifatom*, and *exprtype* as they appear in the rules of inference so that they regard ϕ as an atom of type ρ . (Specifically change *exprtype* by the addition of $x = \phi \rightarrow \rho$ at the beginning of the main conditional. Also if ϕ is a predicate constant, add $x = \phi$ to the disjunction in *Pfatom*. If ϕ is an individual function constant, add $x = \phi$ to the disjunction in *Ifatom*.) (Among other things, these changes allow the substitution of ϕ for the proper function variables.) Second, one adds a single axiom to characterize ϕ . It must be a formula of form $\phi(\xi_1, \xi_2, \dots, \xi_n) \equiv \alpha$ if ϕ is a predicate function constant, of form $\phi(\xi_1, \xi_2, \dots, \xi_n) = \alpha$ if ϕ is an individual function constant, where the ξ_i 's are variables, and they are the only variables free in α . We shall always insist that ϕ occur somewhere in α . For example, if one wants ϕ to stand for the LISP function *maplistcar*, one adds the axiom

$$\phi(f,x) = [atom(x) \rightarrow x ; \oplus \rightarrow f(a(x)) * \phi(f,d(x))] \quad . \quad \text{If one makes}$$

sure that the axiom holds when ϕ is given the desired interpretation, then no inconsistencies are introduced. If it also holds when ϕ has other interpretations, no problems arise; but if it does not hold, whatever the interpretation of ϕ , then an inconsistency might be introduced. Thus, if instead of the above axiom one adds the axiom $\phi(f,x) = \phi(f,x)$ no inconsistencies are introduced. But if one adds $\phi(f,x) = \phi(f,x) * \emptyset$, an inconsistent system results.

By virtue of Rule 20, the addition of axiom $\phi(\xi_1, \xi_2, \dots, \xi_n) \equiv \alpha$ or $\phi(\xi_1, \xi_2, \dots, \xi_n) = \alpha$ allows one to substitute α for $\phi(\eta_1, \eta_2, \dots, \eta_n)$

where α' is derived from α by simultaneously substituting the $\eta_1, \eta_2, \dots, \eta_n$ for $\xi_1, \xi_2, \dots, \xi_n$ (as long as the substitution is not inside a function expression). And Rule 6 allows one to substitute $((\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha), \eta_1, \eta_2, \dots, \eta_n)$ for α' (in the same situations). Thus one can directly substitute $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha)$ for ϕ any place ϕ occurs (outside function expressions) other than as a functional argument. We can regard ϕ and $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha)$ as alternative and interchangeable names for the same function.

Suppose we were to allow interchangeability of ϕ and $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha)$ everywhere outside function expressions, even as functional arguments. This would introduce no inconsistencies since it would only mean, in effect, that we have two names for the same function which, as we have seen above, are already interchangeable whenever it matters, i.e., whenever they are applied to arguments.

We can achieve this effect by using, instead of the atom ϕ , the expression (label, π , $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha'')$) everywhere --- where π is a variable of the same type as ϕ not occurring in α , and α'' can be derived from α by substitution of π for ϕ . Let us call this label expression ϕ' . It is the analogue of the atom ϕ . ϕ' has no free variable and behaves just like ϕ did, in that it can be substituted for function variables by virtue of Rule 5 and the characterizing axiom $\phi'(\xi_1, \xi_2, \dots, \xi_n) \equiv \alpha''$ or $\phi'(\xi_1, \xi_2, \dots, \xi_n) = \alpha''$ (where α'' can be derived from α by substitution of ϕ' for π). Furthermore, we can substitute $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha''')$ for ϕ' anywhere outside of a function expression by virtue of Rule 7.

Thus ϕ' behaves just as ϕ did (i.e., any proof using ϕ' can be converted into a similar proof using ϕ simply by replacing all occurrences of $(\lambda (\xi_1, \xi_2, \dots, \xi_n) \alpha''')$ and ϕ' in the proof by ϕ), with the one addition we wanted and the further addition

that by virtue of Rule 3, we can substitute for bound variables in ϕ' .

Let us say two expressions are almost identical if they are identical or if they differ only by changes of bound variables. Any theorem δ in the system with ϕ has a counterpart δ' in the system with ϕ' where δ' is derived from δ by replacing all occurrences of expressions almost identical to $(\lambda(\xi_1, \xi_2, \dots, \xi_n) \alpha''')$ with ϕ and then replacing all remaining occurrences of expressions almost identical to ϕ' with ϕ . Thus, we have introduced no inconsistency and have added, in effect, a function constant without adding to the alphabet.

We can introduce any label expression in this way by introducing its characteristic axiom as a new axiom, so long as the characteristic axiom holds under the intended interpretation for the label expression. The result will be a new system different from W1 but still consistent. Consider the label expressions whose abbreviations are given in Tables 3 and 6. It is easy to see that the characteristic axiom of each of these label expressions holds if the label expression is interpreted as the function defined by the LISP program that it resembles. Thus, the characteristic axioms of these label expressions may be added to the axiom set without destroying consistency.

Actually, since the characteristic axioms are needed only by Rule 5, and not by Rule 7, any axiom containing the label expression will do as well as the characteristic axiom so long as the label expression occurs in it and so long as it is true under the intended interpretation. We have added axiom 18 to serve for all the label expressions in Tables 3 and 6.

The characteristic axioms now appear as theorems. E.g. consider a ϕ' from these tables which is an individual function expression:

- | | | |
|-----|---|--------------------------|
| (1) | $x = x$ | axiom |
| (2) | $f(\xi_1, \xi_2, \dots, \xi_n) = f(\xi_1, \xi_2, \dots, \xi_n)$ | from (1) by Rule 4 |
| (3) | $\Theta \supset Pfstep(x)$ | axiom |
| (4) | $\phi'(\xi_1, \xi_2, \dots, \xi_n) = \phi'(\xi_1, \xi_2, \dots, \xi_n)$ | from (2) & (3) by Rule 5 |
| (5) | $\phi'(\xi_1, \xi_2, \dots, \xi_n) =$
$((\lambda(\xi_1, \xi_2, \dots, \xi_n)\alpha'''), \xi_1, \xi_2, \dots, \xi_n)$ | from (4) by Rule 7 |
| (6) | $\phi'(\xi_1, \xi_2, \dots, \xi_n) = \alpha''''$ | from (5) by Rule 6 |

With axiom 18 added, the label expressions abbreviated in Tables 3 and 6 are in the initial set, and only expressions almost identical to them are generatable. Since axiom 17 holds under our intended interpretation for \leq , addition of this axiom can add no inconsistencies. Note that none of the label expressions appearing in theorems of W2 have any free variables.

Consistency of W3

The System W3 is just like the limited system except for the addition of Rule 20 and the changing of Rules 4 and 5 so as not to allow substitution for variables free in label expressions.

We shall show consistency of W3 relative to W2 by showing that a proof of Θ in W3 may be transformed into a proof of Θ in W2. Suppose $\alpha_1, \alpha_2, \dots, \alpha_n$ is a proof of Θ in W3. We shall transform the proof successively. After each transformation we will still have a proof of Θ in W3. After the last transformation we will have a proof of Θ in W2.

The first transformation is accomplished as follows:

First, divide into equivalence classes the set of all label expressions appearing in $\alpha_1, \alpha_2, \dots, \alpha_n$ by putting two label expressions in the same class if and only if they are almost identical (i.e., identical but for

changes in bound variables). Let Σ be the set of all such equivalence classes which don't contain a member of the initial set (i.e., don't contain a label expression occurring in axiom 18). If Σ is empty then our proof can be trivially converted into a proof of Θ in W2, by replacing any line derived by Rule 16 or 17 (Such a line contains only label expressions almost identical to those in Axiom 18.) by a derivation of that line from Axiom 18 by P.C. and Rules 5^u and 3. Otherwise, to each member Θ of Σ we attach an integer called the rating of Θ . The rating of Θ is defined to be the smallest integer I such that a label expression in Θ occurs in α_I . (No two members of Σ have the same rating because each use of Rule 16 or 17 generates only one new label expression — because of the $\frac{1}{2}$ in the statement of the rule.) Let Π be the member of Σ which has the largest rating. Then no member of Π occurs in our proof as a proper subexpression of another label expression.

Now consider the line α_I , where I is the rating of Π . Then α_I must have been derived via Rule 16 or 17 from some line, say α_j . α_j must be of form $\phi(\xi_1, \xi_2, \dots, \xi_m) \equiv \beta$ or $\phi(\xi_1, \xi_2, \dots, \xi_m) = \beta$. Let $\eta_1, \eta_2, \dots, \eta_k$ be the variables free in ϕ whose free occurrences in α_j are all in subexpressions of form ϕ . Let $\zeta_1, \zeta_2, \dots, \zeta_k$ be a sequence of variables occurring nowhere in $\alpha_1, \alpha_2, \dots, \alpha_n$ with ζ_i and η_i of same type for all i . Let α_j' be the result of simultaneous substitution of $\zeta_1, \zeta_2, \dots, \zeta_k$ for free occurrences of $\eta_1, \eta_2, \dots, \eta_k$ in α_j . For any i , let α_i'' be the result of substituting the $\zeta_1, \zeta_2, \dots, \zeta_k$ for all occurrences of $\eta_1, \eta_2, \dots, \eta_k$ in α_i . Then $\alpha_1'', \alpha_2'', \dots, \alpha_j''$ can be easily converted into a proof of α_j'' in W3 by prefixing certain lines. For each α_i ($1 \leq i \leq j$) which is an axiom, we prefix lines deriving α_i'' from α_i via Rules 3 and 4.

If $\delta_1, \delta_2, \dots, \delta_h$ are the lines we prefix to do this, then

$\delta_1, \delta_2, \dots, \delta_h, \alpha_1', \alpha_2', \dots, \alpha_j', \epsilon_1, \epsilon_2, \dots, \epsilon_\ell, \alpha_j'$ is a proof of α_j' where ϵ_i is derived from ϵ_{i-1} (and ϵ_1 from α_j' and α_j' from ϵ_ℓ) by Rule 3. We insert this proof into $\alpha_1, \alpha_2, \dots, \alpha_n$ immediately after the line α_j . Note that this addition does not increase the number of classes of label expressions appearing in the proof. Now α_j' is of form

$$\phi'(\xi_1, \xi_2, \dots, \xi_n) \equiv \beta \quad \text{or} \quad \phi'(\xi_1, \xi_2, \dots, \xi_n) = \beta. \quad \text{Also, all numbers of } \Pi \text{ occur in the proof only after } \alpha_j', \text{ i.e., in lines } \alpha_{j+1}, \alpha_{j+2}, \dots, \alpha_n.$$

Now, a member of Π is of form (label, π, γ). If γ' is the result of replacing all free occurrences of π , in γ with (label, π, γ), then we call γ' the expanded form of (label, π, γ). Two label expressions with the same expanded form must be the same label expression, for otherwise they would each occur as a proper subexpression of the other. Let Π' be the set of expanded forms of members of Π . Let α_i' , for $j+1 \leq i \leq n$, be the formula formed first by replacing all occurrences of members of Π' in α_i with ϕ' , and then replacing all remaining occurrences of members of Π by ϕ' . Note, in the replacement no new variables become bound since all free variables in ϕ' which do not occur free in the element of Π' or Π being substituted for, do not occur in α_i . Now we can easily insert lines in the sequence $\alpha_1, \alpha_2, \dots, \alpha_j, \delta_1, \delta_2, \dots, \delta_h, \alpha_1', \alpha_2', \dots, \alpha_j', \epsilon_1, \epsilon_2, \dots, \epsilon_\ell, \alpha_j', \alpha_{j+1}', \alpha_{j+2}', \dots, \alpha_n'$ to make it a proof in W3 in which no member of Π appears. It is clearly a proof up to the α_j' .

Suppose α_i' is some line after α_j' .

If α_i was an axiom or was derived via any rule but 6, 16, 17, we add no lines. In such a case, either α_i' is α_i or α_i was derived from α_K or from α_K and α_L via Rule 1, 2, 3, 4^u , 5^u , 7, 12, or 20. Call the rule used ρ . In such a case we must have $K > j$ or (in the case of Rule 1, 5^u , or 20) $K > j$ and $L > j$. Then α_i' is derivable directly from α_K' or from α_K' and α_L' via Rule ρ . (Note, if we were using rules 4 and 5 instead of 4^u and 5^u this would fail.)

If α_i was derived from α_K via Rule 6, then we must examine the value of the parameter y in the Rule 6 application. It must be of form

$$\theta(\eta_1, \eta_2, \dots, \eta_m) .$$

(A) If θ is not a member of Π' , then we add nothing, and α_i' is α_i or else $K > j$ and α_i' is derivable from α_K' via Rule 6, since any member of Π or Π' occurring inside θ can contain no dummy variable free (because in W3, as in the limited system, and as in our whole system, no variable in a theorem occurs bound inside a label expression, where the binder binding it is outside the label expression —see Section 2.1.4.4) and hence the members of Π and Π' remain unchanged when the η_i 's are substituted for the dummy variables. Thus the Rule 6 application still works when ϕ' is substituted for members of Π' and Π .

(B) If θ is a member of Π' , then we intercalate lines deriving, from α_j' , a theorem α_j'' of form $\phi'(\eta_1, \eta_2, \dots, \eta_n) \equiv \beta''$ or $\phi'(\eta_1, \eta_2, \dots, \eta_n) = \beta''$, by substitution of the η_i 's for the ξ_i 's in α_j' . Then α_i' can be derived from α_K' and α_j'' via Rule 20.

If α_i was derived from α_K via Rule 16, then α_i is of form

$\Theta \supset \theta(\zeta_1, \zeta_2, \dots, \zeta_n)$. If θ is not in Π then neither α_i nor α_K contain an occurrence of a member of Π and α_i' is α_i and is still derivable via Rule 16 just as before. Hence we need add nothing. If θ is in Π then α_i' is $\Theta \supset \phi'(\zeta_1, \zeta_2, \dots, \zeta_n)$ whose proof, via Rules 4 and 5 from $\Theta \supset p$, α_j' , and α_K , we can easily intercalate.

If α_i was derived from α_K via Rule 17, we proceed as in the Rule 16 case.

After all the required intercalations, we end up with a proof of Θ in $W3$ which is much like our original proof. But now all members of Π have been eliminated and yet we have added no new equivalence classes of label expressions (no new members of Σ). The set Σ for this new proof has one fewer member than the set Σ for the old proof.

We can now transform the new proof in the same way we transformed the old proof, giving us a still smaller Σ . We can repeat this until we get an empty Σ , and the proof thus arrived at can be trivially converted (as we said above) into a proof of Θ in $W2$.

Hence if Θ is provable in $W3$, it is provable in $W2$. Then, since $W2$ is consistent, $W3$ is consistent.

Consistency of $W3'$

The system $W3'$ is identical to $W3$ except for change of Rule 5^u to $5'^u$. At first sight it might appear that this change adds some power to the rules, but it is clear upon reflection that any proof in $W3'$ can easily be converted into a proof of the same theorem in $W3$, simply by replacing each step which uses Rule $5'^u$ by the proper sequence of steps using Rules 4^u and 5^u . Hence the theorem sets of $W3$ and $W3'$ are identical, and $W3'$ is consistent because $W3$ is consistent.

Consistency of W4'

W4' differs from W3' in the substitution of Rules 4 and 5' for 4^u and 5^u. We shall show the consistency of W4' by showing that W4' and W3' have the same theorem set. We shall do this by showing that a proof of α in W4' can be transformed into a proof of α in W3'. We shall, then, be considering transformations of proofs in W4'.

We shall begin by considering transformations of a special kind of W4' proof which we shall call a single substitution proof. A single substitution proof is a proof in W4' which becomes a proof in W3' upon deletion of its last line. We shall first show that we can transform any single substitution proof into a W3' proof which contains all the lines of the single substitution proof. The transformation will be in two steps, the first step being what we shall call preprocessing. To see what preprocessing is, we need to discuss some transformations of proofs in W3'.

If $\alpha_1, \alpha_2, \dots, \alpha_n$ is a proof in W3' and η and ζ are variables of the same type, ζ not occurring anywhere in the proof, and if α_i' is the result of replacing η by ζ in all bound occurrences in α_i , then there is a proof $\beta_1, \beta_2, \dots, \beta_m$ in W3' in which each of the α_i' appear as lines and in which η never appears bound except perhaps in formulae which are almost identical to axioms (i.e., identical but for change of bound variable). We construct the proof $\beta_1, \beta_2, \dots, \beta_m$ by intercalating lines in the pseudo proof $\alpha_1', \alpha_2', \dots, \alpha_n'$. Now $\alpha_1', \alpha_2', \dots, \alpha_n'$ is almost a proof as it stands.

Certain of the steps, however, are "illegal." I. e., they would be legitimate steps in W3' if it weren't for an unusual change of bound variables. We shall show how to add lines to successively reduce the number of "illegal" steps.

Let α_j' be the consequent of one of the "illegal" steps. It is not hard to convince ourselves that α_j must either be an axiom or it must be the result of application of Rules 2, 16, or 17. If α_j is an axiom, then we insert before α_j' in the pseudo proof, steps deriving α_j' from α_j by successive applications of Rule 3. If α_j is the result of an application of Rule 2, 16, or 17, then we modify $\alpha_1, \alpha_2, \dots, \alpha_j$ by changing all η into ζ (and then adding lines so that the axioms with ζ substituted for η are legitimately derived from the real axioms via Rules 3 and 4). The resulting sequence is inserted into the pseudo-proof before α_j' . This puts, before α_j' , a theorem identical to α_j' except that free η 's are replaced by ζ . This theorem will be identical to α_j' if α_j was originally generated via Rule 2. Otherwise α_j' may be legitimately derived from this theorem via Rule 4, since none of the free η 's in α_j' are inside a label expression, α_j' being in the special form taken by the results of Rules 16 or 17. This eliminates one "illegal" step. We proceed in this way to eliminate each "illegal" step. This finally gives us a proof in $W3'$ in which η does not appear bound, and in which each α_i' is present.

Suppose $\alpha_1, \alpha_2, \dots, \alpha_n$ is a proof in $W3'$; we can convert this proof with respect to an expression β by successively transforming the proof as above such that for each variable free in β , this variable does not appear bound in the converted proof, and such that for each α_i there is a line in the converted proof almost identical to α_i (i.e., identical to α_i but for a possible change of bound variables).

We are now ready to discuss single substitution proofs. A proof $\nu_1, \nu_2, \dots, \nu_k$ in $W4'$ is called a single substitution proof if $\nu_1, \nu_2, \dots, \nu_{k-1}$ is a proof in $W3'$. If the single substitution proof is not itself a proof in $W3'$ then ν_k must have been derived by substitution of an expression β for a free variable ξ via Rule 4 or 5'.

We shall first show we can transform all such single substitution proofs into proofs in $W3'$. We do this in two steps.

The first step is preprocessing, which is accomplished as follows:

We first convert v_1, v_2, \dots, v_{k-1} with respect to β , giving $\alpha_1, \alpha_2, \dots, \alpha_{n-1}$. Then there must be a formula α_n almost identical to v_k such that $\alpha_1, \alpha_2, \dots, \alpha_n$ is a single substitution proof, and such that α_n is derived by substitution of β' for ξ , where β' is identical to β except for certain changes in bound variables. Pick one such α_n . Then $\alpha_1, \alpha_2, \dots, \alpha_n$ is a preprocessed form of v_1, v_2, \dots, v_k .

$\alpha_1, \alpha_2, \dots, \alpha_n$ is a single substitution proof of a special kind, for not only is its last step derived by substitution of an expression β' for a free variable ξ (as is the case for all single substitution proofs), but variables free in β' occur bound nowhere in $\alpha_1, \alpha_2, \dots, \alpha_n$. Such a single substitution proof is called a preprocessed single substitution proof. (A single substitution proof which is already a proof in $W3'$ is also called a preprocessed single substitution proof.)

If α_n in such a preprocessed single substitution proof is derived by Rule 4 or 5' from line α_i by substitution for a variable free in a label expression in α_i then we say the rating of the proof is i . Otherwise we say the rating is 1. (Note: the first line of a proof, being an axiom, contains no label expressions with free variables.)

We shall now show we can transform every preprocessed single substitution proof $\alpha_1, \alpha_2, \dots, \alpha_n$ into a proof in $W3'$ in which each α_i (for $1 \leq i \leq n$) occurs as a line. We shall do this inductively. Proofs of rating 1 can be transformed trivially. Suppose we know how to transform all preprocessed single substitution proofs of rating less than j . We shall show how to transform an arbitrary single substitution proof of rating j .

Suppose $\alpha_1, \alpha_2, \dots, \alpha_n$ is such an arbitrary proof. Since the proof has rating j we know α_n is derived via Rule 4 or 5' by substitution of β for ξ in the line α_j , where ξ occurs free in α_j inside a label expression. Hence α_j is not an axiom nor is it a theorem which is almost identical to an axiom, since such formulae have no variables free inside label expressions. We also know that α_j

wasn't derived by Rule 8, 9, 10, 11, 13, 14, or 15 since theorems from these rules have no labels.

Suppose α_j was derived from α_k ($k < j$) by Rule ρ where Rule ρ is either Rule 2, 3, 6, 7, 12, 16, or 17. In that case let γ be the result of substituting β for ξ in α_k (this substitution is always legal because variables free in β don't occur bound in α_k unless α_j is almost identical to an axiom, and we said this wasn't the case). Consider the proof $\alpha_1, \alpha_2, \dots, \alpha_{n-1}, \gamma$. This is a preprocessed single substitution proof of rating k and by the induction assumption can be converted into a proof $\beta_1, \beta_2, \dots, \beta_m$ ($m \geq n$) in $W3'$ in which γ and each α_i (for $1 \leq i \leq n-1$) occurs as a line. Now α_n can be derived from γ via Rule ρ , where the parameter values are modified by substitution of β for ξ where appropriate. (Some detailed checking is needed to verify this; we shall not reproduce the details here.) Also, γ occurs in $\beta_1, \beta_2, \dots, \beta_m$ so $\beta_1, \beta_2, \dots, \beta_m, \alpha_n$ is our desired converted proof of $\alpha_1, \alpha_2, \dots, \alpha_n$.

We can make a similar argument if α_j was derived from α_k and $\alpha_{k'}$ by Rule 1 or 20. Let γ and γ' be the result of substituting β for ξ in α_k and $\alpha_{k'}$ respectively. Then by our induction assumption we can convert $\alpha_1, \alpha_2, \dots, \alpha_{n-1}, \gamma$ and $\alpha_1, \alpha_2, \dots, \alpha_{n-1}, \gamma'$ into proofs $\beta_1, \beta_2, \dots, \beta_m$ and $\beta_1', \beta_2', \dots, \beta_m'$ in $W3'$. Then $\beta_1, \beta_2, \dots, \beta_m, \beta_1', \beta_2', \dots, \beta_m', \alpha_n$ is our desired converted proof of $\alpha_1, \alpha_2, \dots, \alpha_n$. α_n is derived from γ and γ' by Rule 1 or 20 as is appropriate.

If α_j was derived from α_k and $\alpha_{k'}$ via Rule 5', substituting β'' for η in α_k , then if η is different from ξ we can handle this situation just like the Rule 1 case. If η and ξ are identical, our final proof is $\beta_1', \beta_2', \dots, \beta_m', \alpha_1, \alpha_2, \dots, \alpha_n$ where α_n is derived from α_k and γ' by Rule 5'.

If α_j was derived from α_k via Rule 4, substituting β'' for η in α_k , then we first construct a few lines $\varepsilon_1, \varepsilon_2, \dots, \varepsilon_\ell$ of a $W3'$ proof which contains no labels, with ε_ℓ being a theorem containing β'' . (Since β'' is a simple expression, this is trivial.) Let γ and γ' be the result of replacing free ξ with β' in α_k and ε_ℓ respectively. By the induction assumption we arrive at a proof $\beta_1, \beta_2, \dots, \beta_m$ in $W3'$ containing γ as before. $\beta_1, \beta_2, \dots, \beta_m, \varepsilon_1, \varepsilon_2, \dots, \varepsilon_\ell, \gamma', \alpha_n$ is our converted proof, with α_n derivable from γ and γ' by Rule 5', and γ' derivable from ε_ℓ by substitution of β' for free ξ .

Thus, we have shown inductively that we can convert every preprocessed single substitution proof into a proof in $W3'$.

Suppose $\nu_1, \nu_2, \dots, \nu_K$ is any single substitution proof, with ν_K resulting by substitution of β for ξ . We preprocess $\nu_1, \nu_2, \dots, \nu_K$ to get a preprocessed single substitution proof $\alpha_1, \alpha_2, \dots, \alpha_n$ where α_n is almost identical to ν_K . Convert $\alpha_1, \alpha_2, \dots, \alpha_n$ into a proof $\beta_1, \beta_2, \dots, \beta_m$ in $W3'$ in the manner we have shown. Let $\alpha_n, \epsilon_1, \epsilon_2, \dots, \epsilon_h, \nu_K$ be a derivation of ν_K from α_n by successive changes of bound variables. Since α_n is among $\beta_1, \beta_2, \dots, \beta_m$, the following is a proof in $W3'$:

$$\nu_1, \nu_2, \dots, \nu_{K-1}, \beta_1, \beta_2, \dots, \beta_m, \epsilon_1, \epsilon_2, \dots, \epsilon_h, \nu_K$$

Thus we have shown that we can convert all single substitution proofs into proofs in $W3'$ containing all the lines of the single substitution proof.

This means we can convert any proof in $W4'$ into a proof in $W3'$ as follows: Suppose $\alpha_1, \alpha_2, \dots, \alpha_n$ is a proof in $W4'$ and suppose ℓ of the lines are derived by substitution for a variable free inside a label expression. Suppose α_j is the first of these undesirable lines. Then $\alpha_1, \alpha_2, \dots, \alpha_j$ is a single substitution proof which we can convert into a proof $\beta_1, \beta_2, \dots, \beta_m$ in $W3'$. Then $\beta_1, \beta_2, \dots, \beta_m, \alpha_{j+1}, \alpha_{j+2}, \dots, \alpha_n$ is a proof in $W4'$ with only $\ell - 1$ undesirable lines. Proceeding in this way, we eliminate all undesirable lines, giving us a proof in $W3'$.

Since all of the original α_i 's appear in the converted proof, anything provable in $W4'$ is provable in $W3'$; $W4'$ and $W3'$ have the same theorem set. Therefore, $W4'$ is consistent since $W3'$ is.

Consistency of $W4$

$W4$ differs from $W4'$ in the elimination of axiom 14' and Rule 20 and the substitution of Rule 5 for Rule 5'. Now Rule 5 is just a special case

of Rule 5'. Thus any proof in $W4$ is a proof in $W4'$. Hence if Θ is provable in $W4$, it is provable in $W4'$. By the consistency of $W4'$, then, $W4$ is consistent. Since $W4$ is the limited system, we have shown the consistency of the limited system as promised.

2.2.1.2 Preservation of Consistency while making identifications Between Object and Meta Levels. (Completion of Consistency argument.) We have shown the consistency of the limited system; we must now show the consistency of our entire system. Our entire system differs from the limited system in the addition of Rules 18 and 19 and the addition of a procedure for adding still more rules.

We shall first show that the addition of Rules 18 and 19 to the limited system will not destroy consistency, will not even add any new theorems to the system's theorem set.

As we said in Section 2.1.4.2, Rules 1-17 may be written as formulae in the object language. (It is not hard to show that T is a generatable function expression.) In fact, so written, they happen to be theorems of the limited system. (This dual nature of rule-expressions is basic to our system: In the object language, these expressions are theorems. In the meta language, they are rules of inference. The exception, Rule 19, is discussed below.)

A completely equivalent form for Rules 1-17 is given in Table 7. A glance at the Table 7 formulation of Rules 1-17 and at the definition of T shows us clearly that the following two meta theorems hold for the limited system.

(1) For any S-expression α , if α is a theorem then $T(\alpha)$ is a theorem.

(2) For any S-expression α , if $T(\alpha)$ is a theorem then α is a theorem.

Thus the addition of Rules 18 and 19 (which merely state these meta-theorems) to the limited system as rules of inference adds no new theorems and so can't destroy consistency.

Are Rules 18 and 19 also theorems of the limited system, the way Rules 1-17 are? We have a recursive procedure to convert, for any α , any proof of α into a proof of $T(\alpha)$. Thus it is not hard to prove Rule 18

as a theorem. There is, however, no recursive procedure which will take us, for any α , from any proof of $T(\alpha)$ to a proof of α . It turns out that Rule 19, as a theorem, is not provable and it is the only rule that is not so provable.

We could increase the set of axioms by adding Rule 19 as an axiom. This would not destroy consistency, but since the T in the new axiom refers to a theorem in the old system and not a theorem in the new system with the axiom added, we just have the same thing to do over again, this time with a T_1 , which means theoremhood in the new system, etc., etc. Can we rewrite Rule 19, replacing T with a function which means provable in the system to which this re-written Rule 19, has been added, thus solving the problem once and for all? Yes, it is possible to so re-write Rule 19; but whether the addition of this re-written rule as an axiom destroys consistency, I do not know.

Now we have shown that the limited system with Rules 18 and 19 added is consistent. Consider the class of theorems in this system of the form $\alpha \supset T(\beta)$. Any theorem in this class may be regarded, as described in Section 2.1.4.2, as the description of a possible rule of inference. Any such rule of inference could be added to our system without destroying consistency, without even adding anything to the set of theorems of the system.

To see why this is so it is only necessary to see how a proof step using this new rule could be replaced by steps using the old rules. Consider a proof step in which the new rule $\alpha \supset T(\beta)$ is used to derive theorem ω from theorems $\gamma_1, \gamma_2, \dots, \gamma_n$ with the parameter values $\mu_1, \mu_2, \dots, \mu_m$ being used for the free variables $\zeta_1, \zeta_2, \dots, \zeta_m$. (See Section 2.1.4.2 for description of how the new rule is applied and for explanation of our terminology.) We shall show how to derive ω without the new rule.

By Rule 18 we can derive theorems $T(\gamma_1)$, $T(\gamma_2)$, ..., $T(\gamma_n)$.

By hypothesis, $\alpha \supset T(\beta)$ is an already proved theorem. We can use Rule 4 to substitute the parameter values for the free variables in this theorem, giving a new theorem $\varepsilon \supset T(\delta)$.

We shall make use of the following meta-theorem.

(A) $(F(x) \wedge \text{oneapl}(x) = y) \supset T(x \equiv y)$ (The proof is by induction on the Gödel number of the expression named by x .) *oneapl* is that function from which *apl* is built by iteration. Its definition is in Table 9, and it is generatable. The above meta-theorem is the meta-theorem from which one would prove the meta-theorem

(B) $(F(x) \wedge \text{apl}(x) = y) \supset T(x \equiv y)$ which we stated in English in Section 2.1.4.2.

Let $oneapl'$ be defined exactly as $oneapl$ is except that

$oneapl'(\text{T}(\text{Y}_i))$ names Ⓢ for all Y_i .

I.e., the definition is:

$$\begin{aligned}
 oneapl'(x) ::= & [x = \text{T}(\text{Y}_1) \vee x = \text{T}(\text{Y}_2) \vee \dots \vee x = \text{T}(\text{Y}_n) \rightarrow \text{Ⓢ}; \\
 & mol(x) \rightarrow x ; a(x) = \text{T} \rightarrow \text{T}(oneapl'(ad(x))) ; \\
 & a(x) = \text{pcond} \vee a(x) = \text{cond} \rightarrow \\
 & \quad [atom(d(x)) \rightarrow x ; \\
 & \quad \quad aad(x) = \text{Ⓢ} \rightarrow adad(x) ; \\
 & \quad \quad aad(x) = \text{Ⓢ} \rightarrow a(x) * dd(x) ; \\
 & \quad \quad \text{Ⓢ} \rightarrow a(x) * (\text{T}(oneapl'(aad(x))) \text{Ⓢ} adad(x) \text{Ⓢ} * dd(x))] ; \\
 & a(x) = \text{Ⓢ} \vee a(x) = \text{=} \vee a(x) = \text{*} \rightarrow \\
 & \quad apltwoatoms(\text{T}(a(x)) \text{Ⓢ} oneapl'(ad(x)) \text{Ⓢ} oneapl'(add(x))) ; \\
 & a(x) = \text{a} \vee a(x) = \text{d} \vee Pfatom(a(x)) \vee Ifatom(a(x)) \rightarrow \\
 & \quad aploneatom(\text{T}(a(x)) \text{Ⓢ} oneapl'(ad(x))) ; \\
 & atom(a(x)) \rightarrow x ; \\
 & aa(x) = \text{λ} \rightarrow Ssffixl(ada(x), d(x), adda(x)) ; \\
 & aa(x) = \text{label} \rightarrow Sffix(ada(x), a(x), adda(x)) * d(x) ; \\
 & \text{Ⓢ} \rightarrow x]
 \end{aligned}$$

Let Σ be a class of well-formed expressions defined by the statement that a well-formed expression μ is in Σ if and only if one of the following holds:

1. μ is of form $T(\gamma_i)$ for one of the γ_i ;
2. μ is of form $(\text{cond}, (\rho_1, \sigma_1), (\rho_2, \sigma_2), \dots, (\rho_k, \sigma_k))$ or of form $(\text{pcond}, (\rho_1, \sigma_1), (\rho_2, \sigma_2), \dots, (\rho_k, \sigma_k))$ where ρ_1 is in Σ ;
3. μ is of form (\supset, ρ, σ) or of form $(=, \rho, \sigma)$ or of form $(*, \rho, \sigma)$ where either ρ or σ are in Σ ;
4. μ is of form (a, σ) or of form (d, σ) where σ is in Σ ;
5. μ is of form (ϕ, σ) where ϕ is a function symbol of our alphabet and σ is in Σ .

Note that each μ in Σ contains certain occurrences of $T(\gamma_i)$'s such that if these occurrences are replaced by \emptyset the resulting expression, which we shall designate as $\hat{\mu}$, is not in Σ , and such that replacement of any

fewer occurrences of $T(\mathcal{Y}_j)$'s results in an expression still in Σ . Note also that if μ is a well-formed formula in Σ then $\mu \equiv \hat{\mu}$ is a theorem. This is because the occurrences replaced are all outside function expressions and not within the scope of any quantifiers, and also because the $T(\mathcal{Y}_j)$'s are theorems.

Now we can prove the meta-theorem

$$(C) \quad (F(x) \wedge \text{oneapl}'(x) = y) \supset T(x \equiv y)$$

from meta-theorem (A). We do this as follows. Suppose μ is the well-formed formula named by x in the statement of meta-theorem (C). Let v be the expression such that $\mathcal{V} = \text{oneapl}'(\mathcal{M})$ holds. Then meta-theorem (C) claims $\mu \equiv v$ is provable. Can we show this? If $\text{oneapl}(\mathcal{M}) = \mathcal{V}$ holds, we get $\mu \equiv v$ by meta-theorem (A). If $\text{oneapl}(\mathcal{M}) = \mathcal{V}$ does not hold, then μ is in Σ , in which case either $\hat{\mu} = \mathcal{V}$ holds or $\text{oneapl}(\hat{\mu}) = \mathcal{V}$ holds. In either of these latter cases we can prove $\mu \equiv v$ from meta-theorem (A) and $\mu \equiv \hat{\mu}$. Thus we have proved meta-theorem (C).

Now return to the theorem $\varepsilon \supset T(\delta)$ which we derived above. We know that repeated application of oneapl' to ε eventually yields \mathcal{O} . (This is because repeated application of oneapl' is exactly what we did when we applied the new rule – see procedure for rule application outlined in Section 2.1.4.2 – and the application of the rule was successful.) Hence by meta-theorem (C) we can prove $\mathcal{O} \equiv \varepsilon$ and hence we can prove $T(\delta)$. Now $\text{apl}(\mathcal{O}) = \mathcal{O}$ holds (since the application of the new rule was successful) so by meta-theorem (B) we can prove $\delta \equiv \mathcal{O}$. From axiom 7b we get $P(\delta) \supset P(\mathcal{O})$ and thus $T(\delta) \supset T(\mathcal{O})$ and $T(\mathcal{O})$. Then we get ω via Rule 19. Thus we have proved ω without the use of the new rule. Thus the addition of the rule $\alpha \supset T(\beta)$ to the set of rules of inference did not add any new theorems to the system and so did not destroy consistency.

Our procedure for adding rules of inference to our system is simply to add, as a rule, any theorem of form $\alpha \supset T(\beta)$ that one wishes to. Our system, then, can be thought of as being identical to the limited system except for the addition both of Rules 18 and 19 and of a procedure for adding an unlimited (but always finite) number of additional rules of inference. We have shown that these additions add nothing to the theorem set. Hence our system is consistent since the limited system is.

2.2.2 Incompleteness. Having shown the consistency of our system, we shall now show some of its other formal properties, specifically those properties related to incompleteness.

We shall need to use the theorem

$$(A) \quad T(y \supset x) \supset T(y \supset \text{apl}(x)) \quad .$$

This is a generalization of the theorem $T(x) \supset T(\text{apl}(x))$ whose tedious proof is sketched in Section 4.10.9. A similar technique gives us a proof of (A).

Now theorem (A) may be used as a new "derived" rule of inference, as explained in Section 2.2.1.2. When we use it this way we shall refer to it as Rule (A).

Now define

$$\varnothing(x) \quad =: \quad Sf(\varnothing, (\text{qu}, x), x) \quad .$$

As an example of the use of Rule (A), consider the following derivation:

(1) $P(Q(f(w))) \supset P(Q(f(w)))$ by P.C.

(2) $\sim T(\varnothing(\sim T(\varnothing(z)))) \supset \sim T(\varnothing(\sim T(\varnothing(z))))$

from (1) by Rules 5 and 4 several times

(3) $\sim T(\varnothing(\sim T(\varnothing(z)))) \supset (T(\sim T(\varnothing(\sim T(\varnothing(z)))))) \supset \text{⊕}$

from (2) by Rule (A)

$$(4) \sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}})) \supset \sim T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}}))$$

from (3) by Rule 6

$$(5) T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}})) \supset T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))$$

from (2) by P.C.

$$(6) T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}})) \supset T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}}))$$

from (5) by Rule (A)

We shall show that our system is incomplete by exhibiting a well-formed formula which is not provable and whose negation is not provable. The formula is

$$(7) T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))$$

whose negation is

$$(8) \sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))$$

Now if (7) is provable then

$$(9) T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}}))$$

follows from (6) and (7) via modus ponens.

From this, (8) follows via Rule 19. Further, if (8) is provable then (9) follows via Rule 18 and (7) follows from (4) and (9) by P.C. Hence if either (7) or (8) is provable, they both are and our system is inconsistent. But we showed our system was consistent and so neither (7) nor (8) is provable. Hence our system is incomplete.

Clearly then (9) is not provable either, nor is its negation

$$(10) \sim T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}}))$$

provable. For if (10) were provable then (8) would follow from (6) and (10) by P.C.

Consider the following statement:

(B) "If α is a well-formed formula then $T(\underbrace{\alpha}_{\text{A}}) \supset \alpha$ is provable."

This statement does not hold for our system. Consider the case when α is (8) above. Then the statement claims we can prove

$$(11) \quad T(\underbrace{\sim T(\emptyset)}_{\text{A}}(\underbrace{\sim T(\emptyset(z))}_{\text{B}})) \supset \sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{C}})) \quad .$$

But if we could prove (11) then from (6) and (11) we could prove (8) by the propositional calculus. Since (8) is not provable, neither is (11).

Note the vast difference between the false statement (B) above and the statements of Rules 18 and 19, either as rules or as formulae in the object language. (As formulae, of course, Rule 18 is a theorem and Rule 19 is not.) The statement (B), in our notation, is

$$F(x) \supset T(\underbrace{T((\text{qu}, x)}_{\text{A}})) \supset x \quad .$$

This is not a theorem and is false under the usual interpretation.

Note: if α is statement (8) then $\sim T(\underbrace{\alpha}_{\text{A}})$ is statement (10). In such a case α is not provable and yet $\sim T(\underbrace{\alpha}_{\text{A}})$ is not provable either. Hence it is clear that although T weakly represents the set of theorems, it does not strongly represent the set of theorems nor does it "express" provability in the sense of Mendelsohn [Mendelsohn, 1964 p. 177 ff].

We shall now show the non-provability of consistency within the system.

We shall show that

$$(12) \quad F(x) \supset (\sim T(x) \vee \sim T(\underbrace{\sim x}_{\text{A}}))$$

is not provable. We first consider the following proof:

$$1. \quad \sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}})) \supset \sim T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{B}}))}_{\text{C}})$$

this is (4) which we proved above

$$2. \quad T(\underbrace{T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}})}_{\text{C}}) \supset T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{D}}))$$

from 1. by propositional calculus and then Rule 18

$$3. \quad T(\underbrace{T(\underbrace{\sim T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{A}}))}_{\text{B}})}_{\text{C}}) \supset T(\underbrace{T(\emptyset(\underbrace{\sim T(\emptyset(z))}_{\text{D}}))}_{\text{E}})$$

from 2. and Rule 1 used as a theorem.

$$4. \quad T(x) \supset T(\underbrace{T(\underbrace{qu, x}_{\text{))}}))$$

Rule 18 used as a theorem.

$$5. \quad T(\underbrace{\sim T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}}) \supset T(\underbrace{T(\underbrace{\sim T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}}))$$

from 4. via Rule 4

$$6. \quad \sim T(\underbrace{T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}}) \supset \sim T(\underbrace{\sim T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}})$$

from 3. and 5. by propositional calculus.

Now if (12) were provable we could prove

$$(13) \quad \sim T(\underbrace{T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}}) \vee \sim T(\underbrace{\sim T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}})$$

from (12) and the clearly provable $F(\underbrace{T(\underbrace{\underbrace{\underbrace{\emptyset(z)}_{\text{))}}))}_{\text{))}})$.

Then from line 6. and (13) we could prove (10) which we know is not provable.

Hence (12) is not provable.

We shall now show that for any well-formed formula α , the formula $\sim T(\underbrace{\alpha}_{\text{))}})$ is not provable. Let α be any well-formed formula. Then we can construct the following proof:

$$1. \quad \sim \forall(x) (F(x) \supset (\sim T(x) \vee \sim T(\underbrace{\sim x}_{\text{))}})) \supset \exists(x) (F(x) \wedge T(x) \wedge T(\underbrace{\sim x}_{\text{))}})$$

by P.C.

$$2. \quad (T(x) \wedge T(x \supset \underbrace{\textcircled{F}})) \supset T(\underbrace{\textcircled{F}})$$

from Rule 1 used as a theorem

$$3. \quad \exists(x) (F(x) \wedge T(x) \wedge T(\underbrace{\sim x}_{\text{))}}) \supset T(\underbrace{\textcircled{F}})$$

from 2. and $T(\underbrace{\sim x}_{\text{))}} \supset T(x \supset \underbrace{\textcircled{F}})$ (this from Rule 6 used as a theorem -- tedious proof)

$$4. \quad T(\underbrace{\textcircled{F} \supset \alpha}_{\text{))}})$$

by propositional calculus and Rules 4 and 5, followed by Rule 18.

(This is where we use the fact that α is a well-formed formula.)

$$5. \quad T(\underbrace{\textcircled{F}}) \supset T(\underbrace{\alpha}_{\text{))}})$$

from 4.

$$6. \quad \sim \forall(x) (F(x) \supset (\sim T(x) \vee \sim T(\underbrace{\sim x}_{\text{))}})) \supset T(\underbrace{\alpha}_{\text{))}})$$

from 1., 3., and 5.

$$7. \sim T(\alpha) \supset \forall(x) (F(x) \supset (\sim T(x) \vee \sim T(\sigma x)))$$

from 6. by P.C.

Hence if $\sim T(\alpha)$ is provable we can derive (12) from line 7 above. Since (12) is not provable, as we have already shown, $\sim T(\alpha)$ is not provable.

Are there any S-expressions σ for which $\sim T(\sigma)$ is provable? Yes. We can show that if σ is not a formula type expression, then $\sim T(\sigma)$ is provable. We do this as follows: Suppose σ is an S-expression which is not a formula type expression. Then since Ftp is a complete recursing function expression, we can derive the following lines:

$$1. Ftp(\sigma) \supset Ftp(\sigma)$$

by P.C. and Rule 5

$$2. Ftp(\sigma) \supset \Phi$$

from 1. by Rule (A)

$$3. T(x) \supset Ftp(x)$$

by the same tedious proof we referred to for proof of $T(x) \supset F(x)$

earlier.

$$4. T(\sigma) \supset Ftp(\sigma)$$

from 3.

$$5. \sim T(\sigma)$$

from 4. and 2. by P.C.

3. IMPLEMENTATION

3.1 Purpose of Section 3. In Section 3 I shall show the existence of a class of interesting machines which utilize the language described in the last section. These machines, when regarded as adaptive theorem provers, do not possess those limitations and obvious shortcomings which have plagued previous adaptive theorem proving machines.

The class of machines discussed here is diverse. Machines which possess identical memory structure may differ in their search algorithms. Machines whose memory structure and search algorithms are essentially the same may differ in the specific reward schemes used.

Now the adaptability and rate of growth of a machine in a particular problem environment depends on the reward scheme. I will not in this paper attempt to discuss and compare various reward schemes for various machines. A meaningful discussion along these lines would have to include a more precise definition and meaningful classification of the various problem environments. This is beyond our reach as yet. It is hoped that expected behavior of machines of the kind discussed here might become the basis for such a classification in the future.

Since demonstrations of growth rate and degree of adaptability must await either this kind of detailed study or actual programming, the current attraction of this class of machines comes chiefly from the interesting hierarchical method of handling heuristics, which bears a resemblance to the techniques of Polya, and which avoids the limitations of previous adaptive methods, and from the fact that comparison with other hierarchical systems (eg. [Holland 1961]) suggests that this class of machines contains members with an acceptably limited growth rate. Although the growth rate may be acceptably limited, the absolute size required is probably very large for interesting members of this class of machines. The construction of these

machines may be beyond the range of present day technology.

Simpler members of the class can be programmed on present day computers. One machine in the class, as a result of a very restricted reward scheme, acts just like Newell Shaw, and Simon's General Problem Solver. [Newell, Shaw, Simon 1961]. I will indicate, by way of example, how this machine operates.

The discussion of such examples will be preceded by a characterization of a subclass of the class of machines. This subclass is large enough to include many of the interesting adaptive machines. The discussion of the adaptive properties will necessarily be imprecise, but it will show the operation of the hierarchy and it will indicate several rather human aspects of the problem solving apparatus. The discussion is not designed to specify the properties of any particular machine, but only to indicate the usefulness of the language described in Section 2.

3.2 Characterization of the Subclass of Machines.

A typical machine in the subclass consists of two parts which I shall call the memory and the effector (or central processor). The memory is merely a storage section in which we store already proved theorems, useful formulae, and other items mentioned below. The effector is a set of algorithms which operate on the memory one after the other. Some of these algorithms are indicated in the following typical sequence of effector operations.

Suppose the user presents the machine with a theorem to be proved. The "insert problem" algorithm places the theorem in the memory at a special spot and returns a pointer to that spot. This pointer is handed to the "*refineproof*" algorithm (the only one which we shall discuss in detail) which alters the memory to produce, inside the memory, a proof of the theorem. Another algorithm reads this proof to the user who then says how much he thinks the proof is worth. A fourth algorithm takes this rating and makes certain

rewards of formulae in the memory on its basis (we shall indicate how this is done). After this, other effector algorithms may juggle rewards within the memory, cause certain formulae to be forgotten, and cause other formulae to be produced on the basis of the rewards.

We shall first discuss the structure of the memory. Then the *refineproof* algorithm will be outlined. After that we will discuss, in general terms, the other algorithms and further possible sophistications.

3.3 Structure of Memory

3.3.1 Basic Plan of the Memory Net.

The memory may be thought of as containing a finite number of entities called memory nets. (Later we will see how to arrange things so that we need only store one net.)

The structure of memory nets was partially explained in Section 1. A memory net may be diagrammed as a set of nodes connected by arrows (called paths). The direction an arrow points is only a reference direction; the algorithms are able to follow a path in the direction of the arrow or in the reverse direction. Some arrows are thick (drawn as a double line) and some thin. Various triangular flags are attached to nodes and paths.

There are three types of nodes: join points, formula nodes, and derivation nodes. These are diagrammed respectively as a dot, a rectangle, and a circle. Each join point is attached to only one outgoing arrow. Arrow heads for arrows coming into join points are not drawn. Arrows coming into join points are thin arrows originating at formula nodes. An arrow leaving a join point is thin and terminates at a derivation node. Thick arrows always originate at a formula node and terminate at a derivation node. Each derivation node has two thin arrows coming in from join points, one thick arrow coming in from a formula node, and one thin arrow leaving to a formula node. (A typical neighborhood of a derivation node is seen in Figure 2,

Section 1.) Each formula node has any number of thin arrows coming in from derivation nodes, any number of thick arrows going out to derivation nodes, and any number of thin arrows going out to join points.

Each formula node contains an S-expression of the language of Section 2.

(The S-expression may be thought of as written inside the rectangle.)

The triangular flags are of three types. The tag type contains a T or H.

The parameter type contains an individual variable of the language of Section 2. The value type contains a number.

Not all entities satisfying the above conditions are memory nets, but it will be easier to state the other conditions if we first define "bug value." "Bug value" can be most easily defined if we consider how a memory net might be stored in a digital computer.

One convenient way of storing a memory net in a digital computer is as a memory structure of the LISP type [McCarthy 1962]. We first convert the memory net and all its contained S-expressions into one huge special S-expression called a net-expression. This is then stored as an S-expression is stored in LISP. If we do this then we can write the algorithms of the effector as LISP programs. (We shall indicate the general outline of a LISP program for the *refineproof* algorithm.) Each algorithm then becomes a LISP function whose arguments are to be memory nets.

Dummy variables used in these functions will be called bugs to distinguish them from the individual variables of the language of the preceding section whose values were normal S-expressions (not net-expressions) rather than nets.

(Definition: A sequence will mean an expression of the form $(\alpha_1, \alpha_2, \dots, \alpha_n)$. The α_i are said to be members of the sequence.)

A net may be written as a net-expression in many ways. One way is to select one node to begin with and write a sequence, the first member of which

is the contents of the node (if it's a formula node), the second a sequence of contents of the various flags of the node, and the others data pertinent to the various arrows entering and leaving that node, data such as direction, thickness, contents of any flags on the arrow, and a sub-S-expression in sequence form which describes the node at the far end of the arrow in the same way that the whole sequence describes the original node. By a recursive argument, we see that the whole sequence gives the structure of the entire memory net seen from the point of view of the originally selected node.

The members of the sequence, except the first two, contain sequences which give the structure of the memory net from the point of view of each of the neighbors of the originally selected node. Hence if one begins with a sequence which describes a net from the point of view of one node, and wishes to derive from it a sequence which describes the same net from the point of view of a neighboring node, one has only to pick the appropriate sub-S-expression of the original sequence. And to return from this new sequence to the original, we once again take the appropriate sub-S-expression.

Notice that the net-expression, as we have described it, is an infinitely deep S-expression. I.e., such a representation as we have described above results in an infinite S-expression when we try to represent any net of two or more connected nodes, because each node is a neighbor of its neighbors. However, by using the LISP functions RPLACA and RPLACD in the proper places we can construct a net-expression that, while finite and with no such redundancies, curls back on itself in such a way that it appears to any LISP function operating on it to be the infinitely deep S-expression we have described above.

Such apparently infinite net-expressions will be said to describe a memory net from the point of view of a particular node. These net-expressions will be named by the arguments to which we apply the algorithms in the effector, and they will be the values over which the bugs range. Hence a bug value is an S-expression which describes a particular net from the point of view of a designated node. Each bug value describes a net and designates a node of that net.

Let us make some informal definitions. Suppose ξ is a bug value. Then let us refer to the net described by ξ as $\boxed{\xi}$. Similarly, let us refer to the node designated by ξ as $\hat{\xi}$. Now certain thin arrows may terminate at $\hat{\xi}$. The origins of these arrows are, then, certain nodes in $\boxed{\xi}$. These certain nodes are called thin-predecessors of $\hat{\xi}$. Now other thin arrows may originate at $\hat{\xi}$. The terminations of these arrows are, then, certain other nodes in $\boxed{\xi}$. These certain other nodes are called thin-successors of $\hat{\xi}$. We similarly define thick-predecessors of $\hat{\xi}$ and thick-successors of $\hat{\xi}$ by replacing the word "thin" by "thick" in the above two definitions.

3.3.2 Some LISP Functions on Bug Values

We shall now mention several LISP functions defined on bug values. The precise definition of these functions depends on the precise method of storing memory nets.

Definition of *derivations* and *derivationsusing*: Suppose ξ is a bug value and $\hat{\xi}$ is a formula node. Then *derivations*($\textcircled{\xi}$) and *derivationsusing*($\textcircled{\xi}$) both name sequences of bug values; each of these bug values describes $\boxed{\xi}$. The members of the sequence named by *derivations*($\textcircled{\xi}$) designate the various thin-predecessors of $\hat{\xi}$. The members of the sequence named by *derivationsusing*($\textcircled{\xi}$) designate the various thick-successors of $\hat{\xi}$.

We will want these sequences to be examined by other LISP functions. In

doing so, it will be important to examine and modify the contents of value-type flags attached to the arrows connecting $\hat{\xi}$ to the nodes designated by the members of these sequences. This means that the arrow in question must be specially marked in each member of the sequences. We will assume this has been done; any number of methods are possible. (One method is to generate, instead of the sequences of bug values, sequences of pairs, each pair consisting of a bug value and a pointer to the relevant flag in the bug value. In this way the net itself is not changed. I shall assume for simplicity that the members of the sequences are the bug values, but in a specific implementation scheme this need not be so.)

Definitions of *containedformula*, *T-tagged*, and *H-tagged*: If ξ is a bug value and $\hat{\xi}$ is a formula node, then *containedformula*($\hat{\xi}$), which we write as $\overline{\hat{\xi}}$, names the S-expression contained in $\hat{\xi}$. *T-tagged* and *H-tagged* are LISP predicate expressions. *T-tagged*($\hat{\xi}$) holds if and only if $\hat{\xi}$ has a triangular flag containing a T. *H-tagged*($\hat{\xi}$) holds if and only if $\hat{\xi}$ has a triangular flag containing an H.

Definitions of *down* and *rule*: Suppose δ is a bug value and $\hat{\delta}$ is a derivation node. Then *down*($\hat{\delta}$) and *rule*($\hat{\delta}$) both name bug values describing δ . The bug value named by *down*($\hat{\delta}$) designates the single thin-successor of $\hat{\delta}$. The bug value named by *rule*($\hat{\delta}$) designates the single thick-predecessor of $\hat{\delta}$.

Another condition on memory nets: Recall that if δ is a bug value and $\hat{\delta}$ a derivative node then there are only two thin-predecessors of $\hat{\delta}$, and that these are both join points. One is called the antecedent node associated with $\hat{\delta}$. The other is called the parameter values node associated with $\hat{\delta}$. They may be told apart as follows. Each arrow terminating at the parameter values node has a parameter-type flag. The parameter-type flags on the various arrows coming into such a join point are all

different from one another. Arrows terminating at the antecedent node have no such flags.

Definition of *antecedents* and *parameters*: Suppose δ is a bug value and $\hat{\delta}$ is a derivation node. Then *antecedents*($\hat{\delta}$) names a sequence of bug values each describing δ . These bug values designate the various thin-predecessors of the antecedent node associated with $\hat{\delta}$. Again, suppose δ is a bug value and $\hat{\delta}$ is a derivation node. Then *parameters*($\hat{\delta}$) names a sequence of pairs, each of the form (η_i, β_i) where each η_i is an individual variable and each β_i is a bug value describing δ . The β_i 's designate the various thin-predecessors of the parameter values node associated with $\hat{\delta}$. For each i , η_i is the variable written on the parameter-type flag attached to the thin arrow which goes from $\hat{\beta}_i$ to the parameter values node associated with $\hat{\delta}$.

The functions we have defined allow us to move from one node to another of a net. We can make statements such as: If x designates a derivation node then $x \in \textit{derivations}(\textit{down}(x))$ holds. (Unless the pair scheme of tagging is used which I mentioned.) In this statement, \in was meant to be the LISP function of that name. However, if we ignore the order of members in a sequence and regard it as a set, then \in can be thought of in the set theoretic sense. We will frequently use such set theoretic abbreviations on sequences the order of whose members is unimportant. So instead of $x * \textit{derivations}(x)$ we may write $\textit{derivations}(x) \cup \{x\}$; we may write $x \subseteq y$ instead of $\textit{andlista}((\lambda(z) z \in y), x)$; etc.

We shall need some more LISP functions which we can define by definition statements in LISP m-notation as described in the LISP 1.5 manual [McCarthy 1962]. However, instead of using exactly the m-notation, we shall use a modified m-notation which is identical to the notation used in

Section 2.1.2 in our definition statements for the function type expressions.

For example, we define the following:

$$pair(x,y) ::= [x = \emptyset \rightarrow \emptyset ; \oplus \rightarrow (\langle \langle a(x) \rangle \rangle a(y) \rangle) * pair(d(x), d(y))] ,$$

$$proj_1(x) ::= maplistcar(a, x) ,$$

$$proj_2(x) ::= maplistcar(ad, x) ,$$

$$find(x,y) ::= [y = \emptyset \rightarrow \emptyset ; aa(y) = x \rightarrow a(y) ; \oplus \rightarrow find(x, d(y))] ,$$

and *variablesin* is defined such that *variablesin* (α) names a sequence of the free variables in the well-formed expression $\overline{\alpha}$. Although the notation here is the same as that of Section 2.1.2, the definition statements here imply the actual definition of a new LISP function, whereas the definition statements of Section 2 were merely summaries of our abbreviation conventions. Note, we use $\overline{\alpha}$ as in Section 2.1.4.1.

3.3.3 Condition on Derivation Nodes (Rules and Heuristics). In

Section 2.1.4.2 we saw that a rule of inference was simply a theorem of form $\alpha \supset T(\beta)$. We specified, in that section, the procedure for applying such a rule of inference. Now a heuristic, in our scheme, is formally a formula type expression (not necessarily a theorem) of form $\alpha \supset T(\beta)$. The procedure for applying a heuristic is the same as the procedure, described in Section 2.1.4.2, for applying a rule of inference. Of course, the usefulness of a heuristic depends not on its form as an expression but rather on its position in a memory net. The significance of its position in a memory net was outlined in Section 1.

In Section 1 we did not have available the notation to discuss the actual form of heuristics and rules of inference. We have now specified that form and specified the application procedure. We shall now review the application procedure and re-state, this time with the aid of the notation we

have developed, the way in which the connections to a particular derivation node reflect an application of a rule of inference or heuristic.

In the rest of Section 3 we shall use the word rule to mean something which is either a rule of inference or heuristic. Thus, a rule of inference is a rule which, as an S-expression, is a theorem.

Consider a particular rule of form $\alpha \supset T(\beta)$. Suppose γ is a sequence of pairs such that all the free individual variables in the rule are contained in $\overline{proj_1(\gamma)}$, and for all ξ , if $(\xi) \in proj_1(\gamma)$, holds then $ad(find((\xi), \gamma))$ names a bug value. Then β can be evaluated as follows: (1) for each individual variable v which is free in β , substitute the S-expression named by $ad(find((v), \gamma))$ for all free occurrences of v in β . (2) Then apply the function apl to the result. This application will terminate in time τ and yield a constant, or it won't. If it does we return the result (call it ρ). If it doesn't we return $\textcircled{\tau}$.

The LISP function that does all this is *result*: If ξ names $\alpha \supset T(\beta)$, η names γ , and ζ names a number such that $\textcircled{\tau} = \phi(\zeta)$ holds for some specified function ϕ , then *result*(ξ, η, ζ) names ρ if the application terminated in time τ and names $\textcircled{\tau}$ otherwise.

In a similar way, we could make the same substitutions in α and apply *apl*. However, here we will be likely to encounter a T. Let us simply keep track of our T encounters in the following way. Each time we try to evaluate $apl((T(\delta)))$ let us return the value $\textcircled{\tau}$, but add the S-expression named by the evaluation of $apl(\delta)$ to a special sequence σ which we keep for the purpose. (In evaluating the $apl(\delta)$ we return $\textcircled{\tau}$ when a $T(\delta')$ is encountered, adding the S-expression named by $apl(\delta')$ to σ , etc. etc.) If this process terminates before the time limit τ and yields a $\textcircled{\tau}$, then return $\textcircled{\sigma}$, otherwise return $\textcircled{\textcircled{\tau}}$. The function which does this is *requiredantecedents*:

If ξ , η and ζ are as before, then $requiredantecedents(\xi, \eta, \zeta)$ names σ if the process terminated in time τ and names $(\textcircled{\sigma})$ otherwise.

Thus we can summarize the idea of a derivation via a rule of inference as follows: If ξ names a theorem of form $\alpha \supset T(\beta)$, γ names a sequence of pairs as above, and τ names a number, and if $andlista(T, requiredantecedents(\xi, \gamma, \tau))$ holds, then $T(result(\xi, \gamma, \tau))$ holds. We have described an application of the rule of inference $\alpha \supset T(\beta)$.

This idea of a derivation is what is to be summarized by a derivation node in the net. Hence we want one more condition to hold on memory nets.

For every bug value δ which designates a derivation node, there exists a number κ such that $result(\overline{rule(\textcircled{\delta})}, parameters(\textcircled{\delta}), \kappa) = \overline{down(\textcircled{\delta})}$ holds and $requiredantecedents(\overline{rule(\textcircled{\delta})}, parameters(\textcircled{\delta}), \kappa) \subseteq$
 $maplistcar(containedformula, antecedents(\textcircled{\delta}))$ holds.

When $\overline{rule(\textcircled{\delta})}$ names a theorem the derivation node summarizes the application of a rule of inference. Otherwise it summarizes the application of a heuristic.

3.3.4 Net Changing Functions

All the functions so far discussed whose range is bug values or sequences of bug values or sequences of pairs of variables and bug values have one feature in common. If ϕ is an expression naming such a function and ξ is a bug value which is a subexpression of $\overline{\phi(\eta_1, \eta_2, \dots, \eta_n)}$ then there exists a bug value ζ and an integer i such that ζ is a subexpression of η_i and $\boxed{\zeta}$ is the same net as $\boxed{\xi}$. In other words, no new memory net is produced by ϕ ; it just produces a new pointer (in the LISP sense) to an old net.

We shall now consider functions which actually produce a new or different memory net. We shall consider later how such a new net is stored. For now, we simply specify its net structure.

The most obvious types of functions, of the sort which produce a new net, are those that erase nodes (and their connections) from an old net, or attach a new node to an old net. We will need both types. We will specify the operation of the second type in some detail. The operation of the first type is no different in principle.

Definition of *constructderivation*: Let us suppose ρ is a bug value; ω is a list of pairs such that $variablesin(\overline{\rho}) \subseteq proj_1(\overline{\omega})$ holds and $proj_2(\overline{\omega})$ names a sequence of bug values; α is a sequence of S-expressions; and ξ is a bug value. Let us write ω as $((\eta_1, \zeta_1), (\eta_2, \zeta_2), \dots, (\eta_n, \zeta_n))$ and write α as $(\beta_1, \beta_2, \dots, \beta_n)$. In the cases we are interested in, all the above bug values describe the same net and they all designate formula nodes. (We could define this for cases of bug values not describing the same net, but it is not necessary.) Thus we could diagram a section of the described net by the solid lines in Figure 9. (I have omitted many possible flags.) Then *constructderivation*($\overline{\rho}, \overline{\omega}, \overline{\alpha}, \overline{\xi}$) names a bug value υ such that $\boxed{\upsilon}$ is the net in Figure 9 with the broken lines added. $\hat{\upsilon}$ is indicated in Figure 9,

as are new nodes $\hat{\gamma}_1, \dots, \hat{\gamma}_m$, where $\overline{\gamma_i} = \beta_i$ holds for all i .

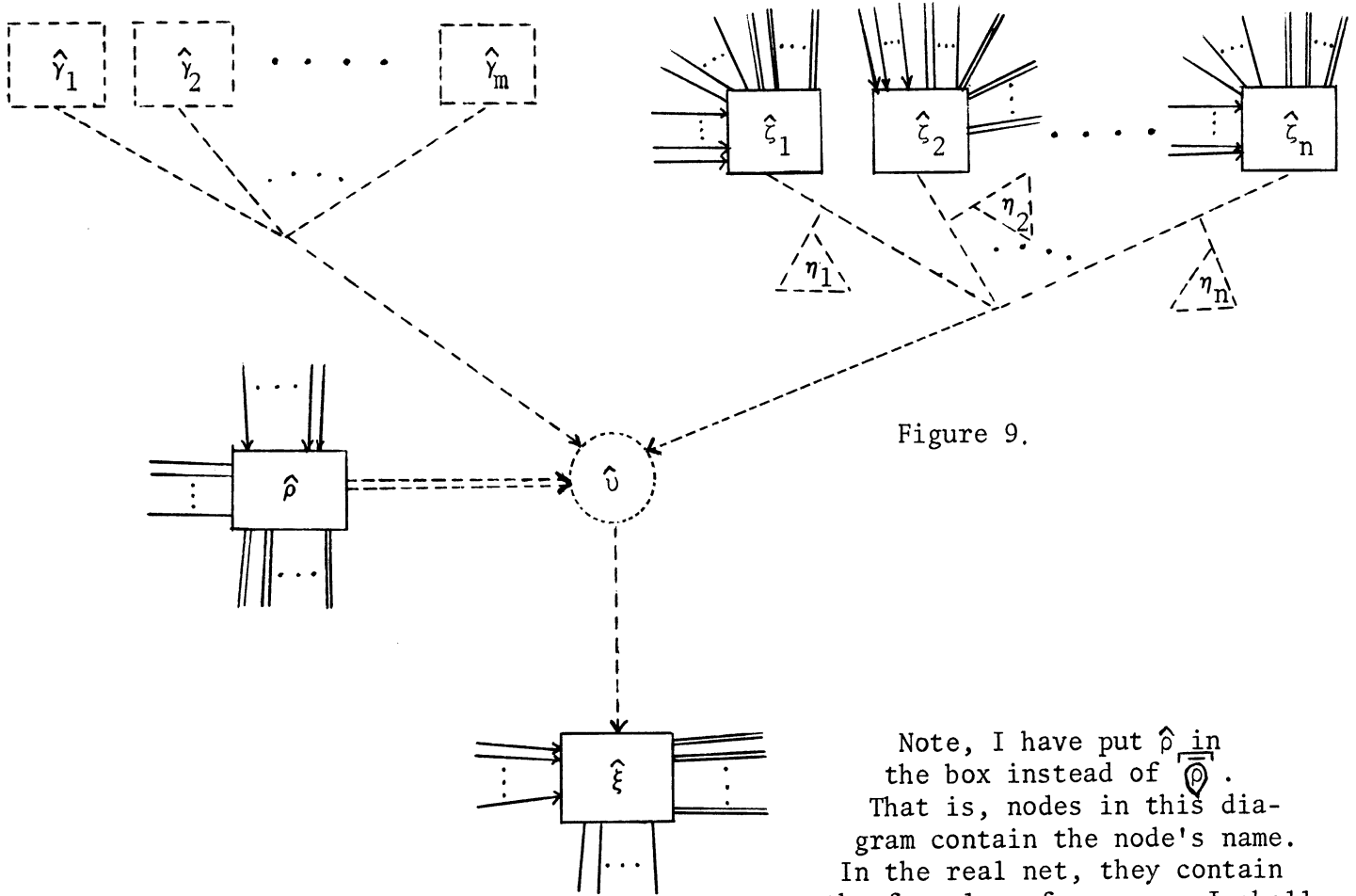


Figure 9.

Note, I have put $\hat{\rho}$ in the box instead of $\overline{\rho}$. That is, nodes in this diagram contain the node's name. In the real net, they contain the formula, of course. I shall follow a similar procedure for nodes (but not for flags) in future diagrams.

Note that once the broken lines have been added, all the labels in the solid boxes are incorrect. For example, $\hat{\xi}$ is a node in the original net, not in the new net. $\overline{\xi}$ is different from \overline{u} . I.e., $down(\overline{u}) = \overline{\xi}$ does not hold, although $\overline{down(\overline{u})} = \overline{\overline{\xi}}$ does hold. Of course, $\overline{\gamma_i}$ is the same as \overline{u} and $\overline{\gamma_i} \in antecedents(\overline{u})$ holds.

Of course, before making such a construction we would want to check to be sure there is a κ such that $result(\overline{\rho}, \omega, \kappa) = \overline{\overline{\xi}}$ holds and $requiredantecedents(\overline{\rho}, \omega, \kappa) \subseteq maplistcar(containedformula, \alpha)$ holds.

Notice that the function *constructderivation* really only constructed the nodes \hat{u} and $\hat{\gamma}_1, \dots, \hat{\gamma}_m$. The others were there to start with. For some purposes we will want to use the similar function, *constructparameterderivation*.

In this case the nodes $\hat{\gamma}_1, \dots, \hat{\gamma}_m$ are available to start with as are $\hat{\rho}$, and $\hat{\zeta}_1, \dots, \hat{\zeta}_n$. The function must construct nodes \hat{u} and $\hat{\xi}$.

Definition of *constructparameterderivation*: Here suppose ρ and ω are as before, and suppose α is a sequence of bug values $(\gamma_1, \dots, \gamma_m)$. As before, assume the nets $\square{\rho}$, $\square{\gamma_i}$, and $\square{\zeta_i}$ are identical for each i . Let χ be an S-expression. Then *constructparameterderivation*($\square{\rho}, \square{\omega}, \square{\alpha}, \square{\chi}$) names a bug value υ , much as before, describing the total net in Figure 9 (though in this case the $\hat{\gamma}_i$'s can have other connections just as the $\hat{\zeta}_i$'s do), where $\overline{\xi} = \overline{\chi}$ holds.

Another sort of function which produces a new net is the *join* function which combines two nodes of a net. We shall give its definition by a general description in English and by an example. Suppose α and β are two bug values, $\square{\alpha}$ is the same as $\square{\beta}$, $\hat{\alpha}$ differs from $\hat{\beta}$, $\hat{\alpha}$ and $\hat{\beta}$ are both formula nodes, and $\overline{\alpha} = \overline{\beta}$ holds. Then *join* changes the net $\square{\alpha}$ into a new net $\square{\gamma}$ by combining the two nodes $\hat{\alpha}$ and $\hat{\beta}$ into one node $\hat{\gamma}$. $\overline{\gamma} = \overline{\alpha}$ holds and all arrows which entered or left $\hat{\alpha}$ or $\hat{\beta}$ in the old net are now diverted so that they enter or leave $\hat{\gamma}$. E.g.:

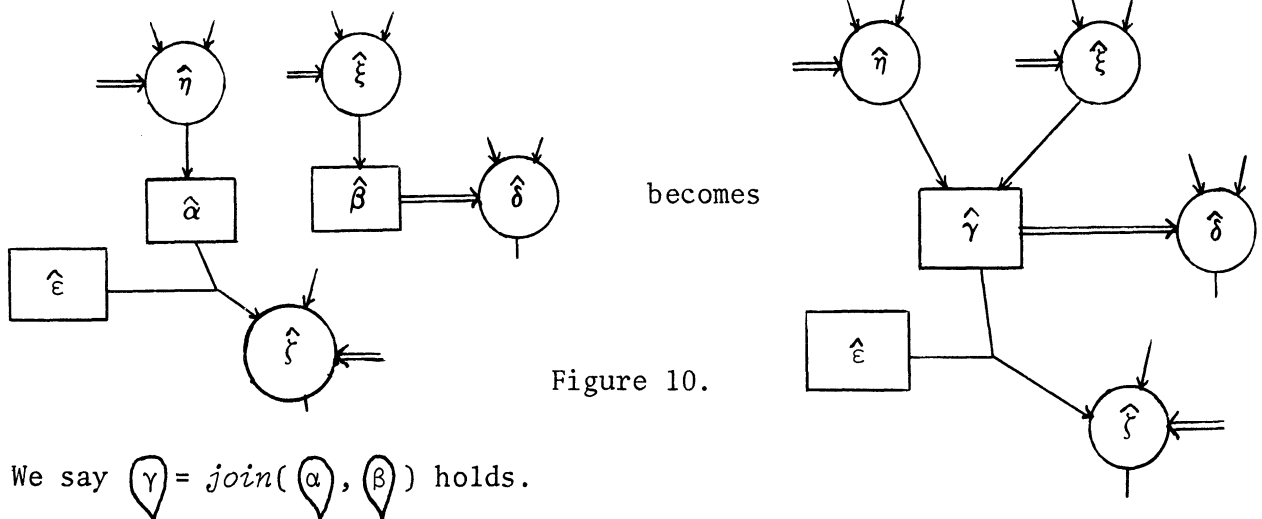


Figure 10.

We say $\overline{\gamma} = \text{join}(\overline{\alpha}, \overline{\beta})$ holds.

3.3.5 Storage of Memory Nets - Difficulties

By way of example, we have indicated three functions (*constructderivation*, *constructparameterderivation*, and *join*) which produce new nets. We now consider an economical way of storing these new nets.

The various algorithms of the effector will be written recursively as LISP programs. The process of following the algorithms will then consist in evaluating an expression of the form $\phi(\eta)$ where ϕ is a LISP function expression (or algorithm) in the effector, and η is a bug value describing a net in the memory. It makes no difference whether we do the evaluation in LISP fashion (with an association list and LISP *eval* function) or in the manner of the evaluations in Section 2 (using substitution via our *apl* function). In either case, we will, during the evaluation, have to keep track of many bug values simultaneously. For example, in the LISP-type evaluation, these bug values will be stored on the association list together with the variables (bugs) which currently have those values. Thus at a given moment we may have to have, in memory, a large number of bug values, just as in normal LISP we have to have a large number of S-expressions. Let us call the set of bug values we must remember at current time, the current bug value set.

Now an S-expression can be thought of as a pointer to a spot in a list structure memory. A bug value can be thought of as a pointer to

a node in a memory net. As long as the members of the current bug value set all describe the same net, the storage problem is easily solved. The memory need only hold that one memory net and the members of the current bug value set are simply pointers to various nodes of that net. Creation of new bug values of the same type causes no problem. Suppose we execute, inside a LISP PROG, $y := \text{down}(x)$ where x and y are bugs and the bug value paired with x on the association list is α . This execution adds y to the

association list, paired with a pointer to a node in the net $\boxed{\alpha}$. We already have the net $\boxed{\alpha}$ in memory, and a pointer to the node $\hat{\alpha}$. That pointer is already paired with x on the association list. The pointer to be paired with y simply points to the neighbor of $\hat{\alpha}$ which is at the end of the thin arrow leaving $\hat{\alpha}$. Though we have created a new bug value, we have not created a new net. The storage problem is simple here. $y := \phi(x)$ (where x and y are bugs) causes no problem, then, unless ϕ actually creates a new net structure.

Suppose now that ϕ creates a new net structure and we try to execute $y := \phi(x)$, where x and y are bugs and the bug value paired with x on the association list is α . (For simplicity we consider a ϕ with a single argument, though the three examples we gave of such a ϕ had 2 or 4 arguments.)

In a normal LISP program such an instruction causes no problems and is handled automatically by the LISP interpreter. It is the fact that our variables are not normal variables, but bugs, that causes trouble. To see why this is so, let's consider the analogue in a normal LISP program (i.e., suppose α is a normal S-expression, not a net expression). Consider, for example, a ϕ which adds something onto α . In a normal LISP program, an example would be $\phi =: (\lambda(u) (\textcircled{z}) * u)$. In executing the instruction $y := \phi(x)$, the LISP interpreter has only to take the pointer paired with x , use it to construct a new pointer to pair with y , and save both the old and new pointers. This works because the value α of x (old value) is to be a subexpression of the value of y (new value).

The situation is quite different if x and y are bugs, and ϕ adds nodes to the net described by α . We add the nodes, create the new net expression, and save the proper pointer to one of the nodes (presumably a new one) to be the bug value for y . But what are we to do with the old

pointer that was the value for x ? New nodes have been added to the net to which it points so it no longer stands for the original bug value α of x . (Looked at another way, the bug value represented by that old pointer now contains a subexpression which is the new bug value for y , designating one of the newly constructed nodes. This certainly was not the case when we started.)

The perpetrators of this unfortunate state of affairs are the pseudo-functions RPLACA and RPLACD hidden inside ϕ . (If there were no LISP pseudo-functions in ϕ , such a thing could never happen, but we need them to make the new nodes into neighbors of their neighbors. Thus any function which constructs new nodes contains such pseudo-functions.) LISP pseudo-functions are notorious for changing the value of a variable behind the variable's back. (see LISP 1.5 manual [McCarthy 1962]).

We would like to be able to write LISP programs for the effector algorithms in the normal recursive LISP style and assume that no bug will change its value unless we tell it to by a statement such as $x := \phi(x)$ or some such. We want to be able to retrieve the old x value if we write

$y := \phi(x)$ followed by $[\Pi(y) \rightarrow \text{return}(y); \Theta \rightarrow \text{return}(\psi(x))]$

where Π names a predicate over bug values and ψ is another function from bug values to bug values. To permit this we must write those functions which change net structure in such a way that the old bug values remain unchanged.

One way of accomplishing this would be to copy the old net, making the required change on the new copy. Thus a new memory net would be added to memory each time an instruction was executed which changed net structure. A memory net would disappear from memory only when all pointers to it had disappeared. Then it would be snapped up by the LISP garbage collector in

the normal LISP fashion. This procedure of duplicating nets has many disadvantages not the least of which is the huge storage space requirement which grows ridiculously for exactly the sorts of procedures we want to reward the system for using.

3.3.6 Patching

Why copy a whole net when we only want to change a tiny part of it? It is more sensible to put a patch over that portion of the net we wish to change.

To illustrate, let us use the example we drew before in Figure 10. Suppose we begin with bug values α and β such that $\overline{\alpha} = \overline{\beta}$ holds and such that $\square{\alpha}$ is the same as $\square{\beta}$ and such that the neighborhoods of $\hat{\alpha}$ and $\hat{\beta}$ are as shown in the left hand portion of Figure 10. Suppose we execute $\gamma := \text{join}(\alpha, \beta)$. Then the resulting net will be (in the neighborhood of $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\gamma}$) as in Figure 11.

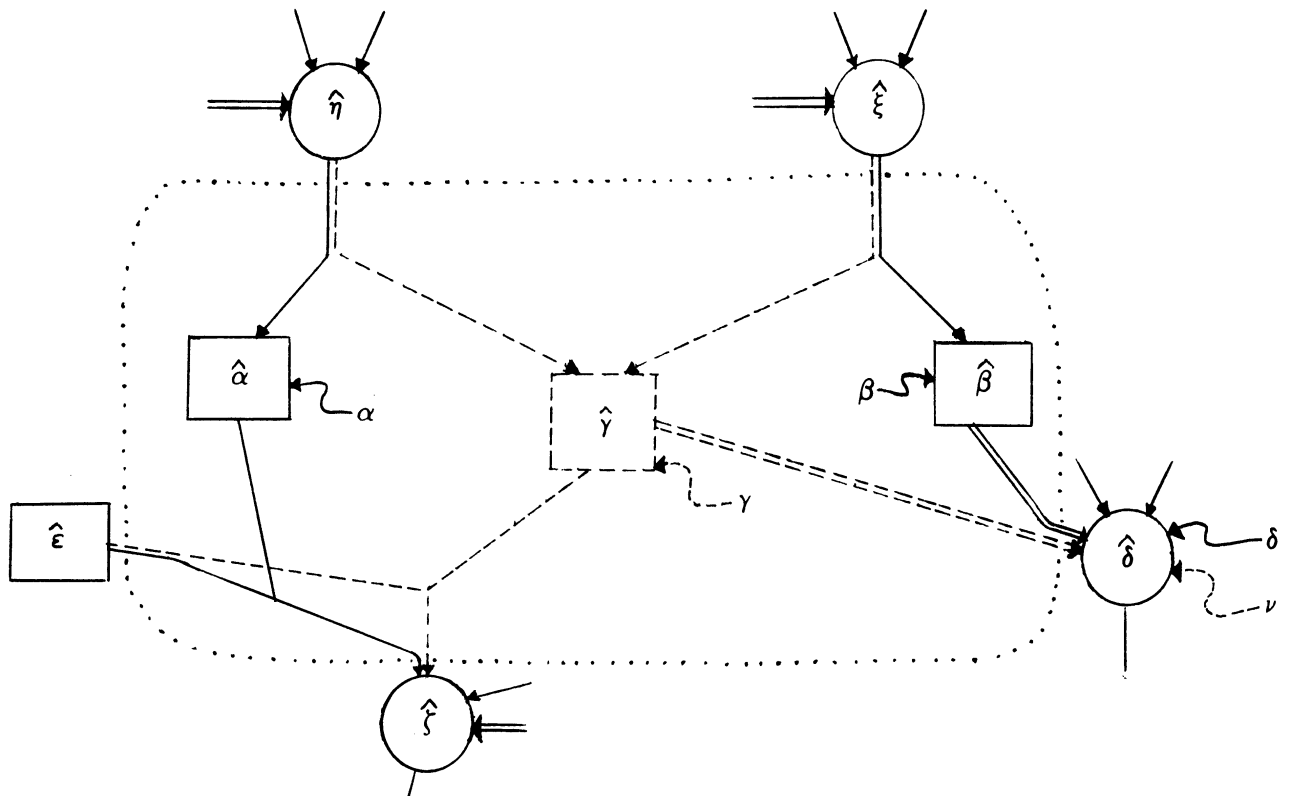


Figure 11.

The paths which have been added are drawn as broken lines. They are to be distinguished from all other lines as they are members of the patch. The outer dotted line indicates the edge of the patch. Any solid line within the edge is covered by the patch. Lines crossing the edge of the patch are double, having a broken and solid component.

Let us regard the bug values α , β , and γ as pointers to nodes in $\boxed{\alpha}$, $\boxed{\beta}$ (or $\boxed{\alpha}$), and $\boxed{\gamma}$ respectively. I have indicated these pointers in Figure 4 by the curvy arrows. The curvy arrow is solid if it is to be regarded as pointing to a node in $\boxed{\alpha}$, and broken if it is to be regarded as pointing to a node in $\boxed{\gamma}$. Now let us indicate the bug values δ and ν defined such that $\delta = a(\text{derivationsusing}(\beta))$ and $\nu = a(\text{derivationsusing}(\gamma))$

hold. The pointers for these bug values are indicated in Figure 4. They seem to point to the same node, but one is solid and one broken so we see δ is meant to point to a node in $\boxed{\alpha}$, and ν is meant to point to a node in $\boxed{\gamma}$. The consequences of this are most clearly seen by attempting to find $\overline{\text{rule}(\delta)}$ and $\overline{\text{rule}(\nu)}$. In either case we start with the node pointed to by the curvy arrow labeled δ or ν . In either case we move backwards along the thick arrow which terminates there. But that arrow has a broken and solid component. In the case of $\overline{\text{rule}(\delta)}$ the curvy arrow was solid so we pick the solid component and move under the patch to $\hat{\beta}$. In the case of $\overline{\text{rule}(\nu)}$ the curvy arrow was broken so we pick the broken component and move up onto the patch to $\hat{\gamma}$. Thus $\beta = \text{rule}(\delta)$ and $\gamma = \text{rule}(\nu)$ hold just as if we had saved two whole nets. Instead of saving two nets with curvy arrows on each, we save one net with a patch, with two sets of curvy arrows (solid and broken); the members of one set "see" the patch, and the members of the other don't "see" the patch as they move over the net.

If, at any time, all bug values whose curvy arrows are broken have disappeared from the association list, then we can erase the patch with impunity. Similarly, if at any time all bug values whose curvy arrows are solid have disappeared from the association list, then we can, with impunity, turn the broken curvy arrows into solid ones, erase the solid stuff under the patch, and make the broken stuff that is on the patch solid.

3.3.7 Tagging and Garbage Collection

Suppose instead of distinguishing the patch by broken vs. solid, we just tag the "broken" component of the double arrows crossing the edge of the patched area with a (1). We similarly tag the "broken" curvy arrows with a (1). We say the patch is tagged with a (1). Thus our patched net stands for two nets, an untagged net and a 1-tagged net.

Now, suppose we want to put on another patch. We just tag the new patch with a different number. The details depend on whether we want this new patch to go on the untagged net or the 1-tagged net. In the first case, the new patch gets tag (2) and if it overlaps the old patch it is placed upon the stuff under the old patch. In the second case the new patch gets tag (1, 2) and is placed upon the stuff written on the old patch. The curvy arrows get the same tag as the patch. Each new patch gets a tagging sequence ending in a number not in any other sequence in the association list. In this way we develop a hierarchy of patches upon patches.

As in LISP, we periodically garbage collect. We examine the association list for redundancies. E.g., suppose all curvy arrows whose tag sequence contains a 3 have a 7 immediately preceding it. In that case the 7 can be dropped and the 3 patch made a part of the net it patched (i.e., of the uppermost patch it patched). The stuff under the 7 patch is then cut loose from the net and is picked up by the regular LISP garbage collector. Converse if 7 appears in no curvy arrow tag sequence, then the 7 patch can be cut loos

Because of the recursive structure of the effector algorithms, the number of patches will drop drastically upon completion of a subroutine. In the examples we shall consider, the number of patches will drop to zero upon completion of a "to prove" problem presented to the machine.

At any given time, then, the memory of a machine consists of a single patched memory net, this being equivalent to many unpatched nets.

3.4 Other Functions Which Change Net Structure

3.4.1 Kinds of Functions to be Discussed

We have yet to specify in more detail certain important algorithms of the effector. These will be built up from primitive functions (on bug values) both of the type which leave net structure unchanged (we have already defined these) and those that change net structure. We have discussed only three of the latter type; we require several more. We will describe how they change net structure; but remember that what is actually accomplished is the creation of a patch as described in Section 3.3.6.

In addition to functions which add nodes to the net like those discussed above, other functions erase nodes. I shall not describe these in detail.

Simpler than either of these kinds of functions are functions which merely change flags. *T-tag* simply attaches a flag containing a T to the node designated by the argument. I.e., $T\text{-tag}(\xi)$ names the bug value η where $\boxed{\eta}$ can be formed from $\boxed{\xi}$ by attaching a new flag with a T to the node $\hat{\xi}$, and where $\hat{\eta}$ is that node to which the new flag has been added. *H-tag* similarly attaches a flag containing an H to the node designated by the argument.

Recall that various value-type flags may be attached here and there to nodes and arrows. We will want functions which read the values on these flags, raise them, and lower them. We will be looking at these values usually in one of three contexts: searching, rewarding, or punishing.

3.4.2 Searching

The simplest search pattern is the search of a sequence of nodes. Suppose $\xi_1, \xi_2, \dots, \xi_n$ are bug values which all describe the same net. Now let us suppose that with each $\hat{\xi}_i$ we can unambiguously associate a number value from some nearby value-type flag. Then we can define a function *bestlist* which makes a probabilistic choice from among the ξ_i on the basis of the size of these values. The bug value returned would then be the ξ_i selected. Actually it will be convenient to raise, at this time, the number on the flag we looked at when we obtained the value to associate with $\hat{\xi}_i$. (I.e., "Unto him that hath shall be given", since a large value means higher probability of being chosen and being chosen raises this very value.)

A more detailed description of the procedure follows. Suppose η is a sequence of bug values all describing the same net, together with a designation, for each bug value, of a value-type flag. (Examples of such a sequence would be $\overline{\text{derivations}(\zeta)}$ or $\overline{\text{derivationsusing}(\zeta)}$ for some bug value ζ . Recall that we decided not to specify in these cases the exact method of distinguishing the value-type flag.) Suppose k is a number. Then $\text{bestlist}(\eta, k)$ names a bug value ξ determined as follows. A probabilistic choice is made among the bug values in η on the basis of the numbers on their associated value-type flags. If none of these numbers is high enough with respect to k , return \emptyset . Otherwise return the chosen bug value and raise the number that was the cause of that bug value being chosen (i.e., raise

the number on the flag). This converts the old bug value into a new one, ξ , which describes a slightly changed net (one flag is changed).

The function *bestlist* is, in a sense, a model for all the search functions we shall use. The principle of the other search functions is the same: select a bug value or pair of bug values from a set of bug values or pairs of bug values; make this selection on the basis of numbers on value-type flags; raise the number on the value-type flag used to select the winning candidate; return \emptyset if the numbers on all the value-type flags are low with respect to a parameter k ; otherwise return the winning candidate, complete with modified flag.

The function *bestproductinlist* is the same as *bestlist* except that the sequence is a sequence of pairs of bug values, each member of the pair with an associated value-type flag. The value of the pair is to be regarded as the product of the numbers on the two value-type flags associated with the members of the pair. Selection is based on this value. The pair selected is modified by raising, in each bug value, the values on both flags used; i.e., the two bug values in the pair returned are modified so that they end up still describing the same net.

Certain parts of our algorithms are designed to handle very bad situations, when no obvious heuristic seems to work. In these cases, searches over a whole net are required in order to find the right heuristic to use. These complicated and, for our purposes here, uninteresting searches can be designed in various ways.

If ξ is a bug value and k a number then *bestnetrule*(ξ, k) names nil or a bug value η constructed as follows: A node $\hat{\rho}$ of ξ is selected probabilistically on the basis of numbers on certain value-type flags attached to the nodes of ξ . (ρ is a bug value and ρ is ξ .) These numbers are, however, not the sole criterion used in the choice. Nodes "closer"

to $\hat{\xi}$ are weighted more likely to be chosen than nodes farther away. The "distance" between a node and $\hat{\xi}$ is calculated with the aid of the numbers on value-type flags attached to arrows along paths connecting the node and $\hat{\xi}$. If the value-type flag on the chosen node $\hat{\rho}$ does not have a high enough number with respect to k , or if ρ is too "far" from $\hat{\xi}$ with respect to k , then $bestnetrule(\hat{\xi}, k)$ names nil. Otherwise ρ is modified to form η by raising the number on the value-type flag attached to $\hat{\rho}$. In this case $bestnetrule(\hat{\xi}, k)$ names η .

Bestnetparameterrule is like *bestnetrule*, but different value-type flags are used to guide the choice of node. Actually the operation of *bestnetparameterrule* is a bit more complicated and will be explained in more detail later, as will *bestnetparametervalue*.

In eliminating patches, etc., in garbage collection, it could happen that the patched net is split into several parts not connected with one another. This creates a small problem. Ad hoc provision will have to be made so that all parts are saved. The three above functions (and other similar ones) will have to be able to look at all saved parts of the net.

3.4.3 Functions on Two Nets

Now we take advantage of the fact that memory is really a single patched net and not several different nets.

It is possible, for example, that, although ξ and η are two different nets, $\hat{\xi}$ and $\hat{\eta}$ are the same node of the patched net. That is, the curvy arrows for ξ and η point to the same node, but they have different tags. Thus we have a one to one correspondence between certain nodes in ξ and certain nodes in η .

Similarly, if ξ and η are two different nets, and $\hat{\xi}$ and $\hat{\eta}$ are not the same node of the patched net, then via such a one to one correspondence

there may be a node $\hat{\zeta}$ of $\boxed{\xi}$ such that $\hat{\zeta}$ and $\hat{\eta}$ are the same node of the patched net. We say $\text{jumpback}(\xi, \eta)$ holds. If no such node $\hat{\zeta}$ exists then $\text{jumpback}(\xi, \eta) = \emptyset$ holds. In other words, the curvy arrow representing $\text{jumpback}(\xi, \eta)$ points where the curvy arrow representing η points, but is tagged as the curvy arrow representing ξ is, whenever such a tag makes sense.

We will need some way of punishing a wrong choice in a search.

Suppose we have selected a certain node $\hat{\zeta}$ in $\boxed{\xi}$ by means of one of the search functions, raised the number τ to π on the appropriate flag \triangleright_{τ} and perhaps even constructed some new nodes. Suppose the net $\boxed{\eta}$ which is the result of all this turns out to be all wrong and we wish to return to $\boxed{\xi}$ and choose a different node. But how are we to keep from choosing $\hat{\zeta}$ again? We want the flag \triangleright_{τ} punished by lowering τ so that there is less chance of picking $\hat{\zeta}$ again. Suppose $\hat{\xi}$ and $\hat{\eta}$ are the same nodes of the patched net in memory. Then $\text{erasepunish}(\xi, \eta)$ names a bug value ω which is just like ξ but with values on value-type flags altered as follows: For each value-type flag \triangleright_{τ} in $\boxed{\xi}$ we find the corresponding one

\triangleright_{π} in \square_{η} . We examine the pair (τ, π) of numbers. (π is 0 if there exists no corresponding flag in \square_{η} .) Set ν equal to $\frac{\tau^2}{\pi}$. (Or perhaps not this exact function. We need a ν equal to $\phi(\tau, \pi)$, where ϕ is a function such that for any numbers τ , π , and μ :

- (1) $\tau = \phi(\tau, \tau)$,
- (2) $\tau > \phi(\tau, \pi) > \phi(\tau, \mu)$ whenever $\tau < \pi < \mu$, and
- (3) $0 \leq \phi(\tau, \pi)$.

Now the value-type flag on \square_{ω} which corresponds to \triangleright_{τ} on \square_{ξ} is to be \triangleright_{ν} . Note that this function can be executed very quickly since actual work is needed only on areas of the patched net where the patches differ for \square_{ξ} and \square_{η} .

punish is just like *erasepunish* except we require the nets \square_{ξ} and \square_{η} to be identical except for values on value-type flags.

reward(\triangleright_{η} , \triangleright_{ξ}) names the same bug value as *erasepunish*(\triangleright_{ξ} , \triangleright_{η}) except that with τ and π defined as before and ν equal to $\phi(\tau, \pi)$, we now require, for all numbers τ , π , and μ ;

- (1) $\tau = \phi(\tau, \tau)$,
- (2) $\pi < \phi(\tau, \pi) < \phi(\tau, \mu)$ whenever $\pi < \tau < \mu$, and
- (3) $0 \leq \phi(\tau, \pi)$.

If some of the flags we are to be rewarding or punishing are to be attached to new derivation nodes, then they had better be added to new nodes by *constructderivation*. I shall not specify just which flags are to go on the new derivation nodes. Suffice it to say here that the definition I gave of *constructderivation* should be modified so that the new node receives the proper value-type flags with the values appropriate monotonic functions of nearby value-type flags.

Similarly, the definition of *join* should be modified so that each of the values on the flags on the new nodes is the sum of the two corresponding values on the corresponding flags on the two old nodes which have been joined.

We shall make frequent use of the term *operate*. If we write an operate statement such as $\phi := \text{operate}(\gamma)$, it means that in the rest of this program (i.e., LISP PROG): if α names a bug value then $\phi(\alpha)$ is to mean *jumpback*(α, γ);

if *atom*(α) holds then $\phi(\alpha)$ is to mean α ; and if α names neither a bug value nor an atom, $\phi(\alpha)$ is to mean $\phi(a(\alpha)) * \phi(d(\alpha))$. We shall find this abbreviation very useful. Any Greek letter may be used on the left of the operate statement. In place of γ I may write any term which names a given bug value.

3.5 LISP Structure of the Effector

The effector may be thought of as a set of LISP programs, each describing a LISP function. The LISP functions are defined over S-expressions and bug values (remember, a bug value is really a specialized S-expression, though we are treating it as something different). These LISP functions are built up from one another in the normal LISP programming fashion. Included in the effector are the following functions: the primitive functions of the LISP 1.5 interpreter; the LISP functions analogous to the function type expressions where abbreviations are given by the definition statements in Sections 4.3, 4.6, 4.7, 4.8, and 4.9; the LISP functions described above in this section (these are defined over bug values). Other functions in the effector are built from these, until finally we arrive at certain functions, such as *refineproof*, which are described recursively in LISP fashion and are the so-called effector algorithms which we have mentioned earlier and which operate on the memory during machine operation. (Since the functions are defined over bug values we must guard carefully against infinite recursion. If α and β are bug values then the evaluation of $\alpha = \beta$ recurses infinitely, at least in some circumstances. I will never evaluate such a form unless it is of the form $\alpha = \emptyset$.)

We are now in a position to describe certain effector algorithms. They will be specified by a LISP program built up from already defined functions (But we will continue to use some of our notation from Section 2: we write * for *cons* and *a* for *car*, etc.,etc.)

3.6 Refining a Proof

3.6.1 The Task of *refineproof*

We shall describe only one of the effector algorithms in some detail, namely *refineproof*. This algorithm is used when the machine is attempting to

produce a proof of an already given formula. Such a problem might arise, for example, if the user presented the machine with a supposed theorem and asked the machine to prove it.

Presented with this sort of a problem, the machine makes a trial proof outline, and then attempts to alternately refine and modify the outline until a complete proof is achieved. The steps of an outline are not necessarily made via rules of inference (i.e. by theorems in rule form). Usually they are made via so-called heuristics. A heuristic may be thought of as a rough approximation to a summary of the rules used in a line of reasoning or sequence of rule of inference applications. How this is so was discussed in Section 1. Technically, a heuristic is merely a formula type expression in the form of a rule (a heuristic need not be a theorem).

In refining the proof outline, a step using a heuristic is replaced by one or more steps which use rules of inference or more detailed heuristics. All these proof outlines will be constructed in the memory net by means of *constructderivation*. We described the refining process in Section 1. We shall give a slightly more formal description of that process here.

The first task of the effector is to take the formula to be proved and construct a simple trial proof outline. The simplest would be a one-step outline. Suppose at this time, the memory holds the single unpatched net $\boxed{\xi}$. The effector searches the net $\boxed{\xi}$ for a heuristic which will yield the required formula in one step. Such a search is similar to those we have discussed and will discuss. It is especially simple if the net contains a node $\hat{\eta}$ such that $\overline{\eta}$ names $\oplus \supset T(x)$, since this heuristic will always yield the required formula. We have but to add the broken line construction below to the net $\boxed{\eta}$.

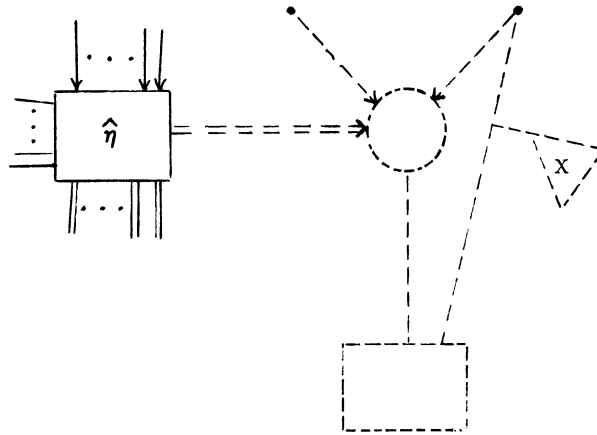


Figure 12.

Note: the new formula node contains the formula to be proved. The antecedent node has no arrows coming into it and needs none.

From this point on we proceed by successive refinements of this proof outline. The refining procedure is accomplished by *refineproof*. The definition of *refineproof* or rather an outline of its definition (along with outlines of definitions of other LISP function names we shall be using) is given in Section 4.11. *Refineproof* makes constant use of T- and H-tags.

3.6.2 Use of T-tags and H-tags.

T-tags and H-tags appear only on formula nodes.

The axioms of the system and the rules of inference which are theorems, are contained in formula nodes which are T-tagged and H-tagged. Certain initial non-theorems (formally only one is required) are H-tagged.

Any other T-tagged node $\hat{\alpha}$ must initially meet the condition that the following holds for α : $\mathcal{H}(y) (y \in \text{derivations}(\alpha) \wedge$

$$(T\text{-tagged}(\text{rule}(y)) \vee \overline{\text{rule}(y)} = \text{T}(\text{T}(\text{qu}, \text{x})) \supset \text{T}(\text{x})) \wedge \text{andlista}(\text{T-tagged}, \text{antecedents}(y)))$$

However, if this condition later ceases to be met due to erasures, the T-tag remains.

Note: We specifically allow the rule to be rule of inference 19, since this is the only rule of inference which is not a theorem. By induction, only

theorems may be T-tagged and hence $\overline{rule(y)}$ must name a rule of inference.

Similarly any other H-tagged node $\hat{\alpha}$ must meet the condition that the following holds for α : $T\text{-tagged}(\alpha) \vee \exists(y) (y \in \text{derivations}(\alpha) \wedge \text{andlista}(\text{H-tagged}, \text{antecedents}(y)))$

If this condition later ceases to be met because of erasures, the H-tag is erased. An H-tag on a node means that the node is the termination of a proof outline (from theorems) via heuristics.

3.6.3 The Task of *prove*

refineproof takes the bug value ξ and the number κ and tries to produce a new net $\overline{\eta}$ by performing constructions on $\overline{\xi}$ such that $\overline{\eta} = \overline{\xi}$ and $T\text{-tagged}(\overline{\eta})$ hold. κ is a positive real number which tells how hard to try (how many unlikely possibilities to try). *refineproof* first checks to see if $\hat{\xi}$ is already T-tagged. If not, it tries to make more precise the various derivations of $\hat{\xi}$. With each derivation it looks at, it first calls itself recursively to see if it can get T-tags on the proper nodes to permit T-tagging of $\hat{\xi}$.

If for a given derivation all this fails, it assumes the rule used was a heuristic and calls *expandheuristic*. This tries to replace the old heuristic-type derivation with a brand new derivation. It searches out new rules, parameter values, and antecedents to use. At worst, the search is random; at best, the old derivation will give a great deal of information as to what the new derivation should be like. Hopefully the rules needed will be nearby in the net (otherwise it was a bad heuristic, or at least *refineproof* is on the wrong track). Perhaps a similar problem has been encountered before (this, of course, is usually the case for any given sub-problem). In that case there will be a record in the net that the heuristic was used to prove theorem χ , and there will also be a record of how χ was

finally proved. It is the job of *expandheuristic* to find the right χ . Then the effector attempts to construct a derivation of $\overline{(\xi)}$ which mimics that of χ .

This construction is done recursively by the function *prove*. It uses a sequence of pairs named by the dummy variable ℓ . This sequence (or set) of ordered pairs may be regarded as a mapping from the χ derivation being mimicked to the $\overline{(\xi)}$ derivation being constructed. Corresponding meta level nodes in the two derivations are identical. All subroutines return a dotted pair (α, β) where α is the current bug value and β is the current ℓ sequence.

Thus, it is the function *prove* which has the task of deciphering the meaning of a heuristic by first trying to mimic a previous use of the heuristic.

3.6.4 Example: A heuristic which is a composition of two rules.

(See Figure 13, solid portion.) Here we see a simple example of a portion of a net illustrating use of a heuristic, namely, $T(\phi(x)) \supset T(\theta(x))$. to prove the formula named by $\theta(\overline{(\xi)})$ i.e., the formula one gets by applying θ to ξ . This formula, as we see, was first "proved" directly from $\overline{\phi(\overline{(\xi)})}$ via the heuristic. Later (that it was later rather than earlier is not obvious from the net) this single step was refined by the creation of the two step derivation via $\overline{\psi(\overline{(\xi)})}$. If this was the first use of the heuristic, the refining process must have been fairly difficult. (Locating the two rules to use would have taken place in *randomprove* via *bestnetrule*($\overline{(\sigma)}, \overline{(\kappa)}$), where $\overline{(\sigma)}$ names our heuristic. *bestnetrule* would have found the right rules by tracing the derivation of the heuristic via $\overline{(\mu)}$.)

However, after such a refined proof has once been achieved, then it is not too difficult to refine a second use of that heuristic in the same way.

Suppose the portion of the net shown solid in Figure 13 is already constructed, and suppose the heuristic is now used to "prove" another theorem $\overline{\theta(\overline{(\zeta)})}$

(See the broken-line derivation of $\theta(\zeta)$ via the heuristic in Figure 13.) Let $\hat{\beta}_3$ be the node containing $\theta(\zeta)$.

We can see how $refineproof(\beta_3, \kappa)$ would evaluate here if we refer to Section 4.11 and the effector function definitions (κ is a number). These definitions are not complete but are merely outlines of the sort of definitions required. (E.g., $\psi(\kappa)$ stands here for the name of a number related to κ , but smaller than κ . I have not specified in these cases how the new number is calculated). (Some obvious improvements in the definitions come immediately to mind, such as diagonalization of sets of searches that are here handled consecutively.)

I have made in these definitions one violent simplifying assumption: that each rule contains only one free variable. This is ridiculous, of course, but allows the definitions to be written much more simply since the searches for parameter values are straightforward searches rather than complicated diagonal ones. The simplification is made only to make the principles of search easier to see, and could not be made in a real machine. In a real machine, these searches would be diagonalized in one of the obvious ways.

Happily, the simplifying assumption just happens to be true for the example of Figure 13. The function *parameters* is changed to *parameter* and returns a pair instead of a sequence of pairs. Hence we can use the definitions in Section 4.11 to clarify the evaluation of $refineproof(\beta_3, \kappa)$.

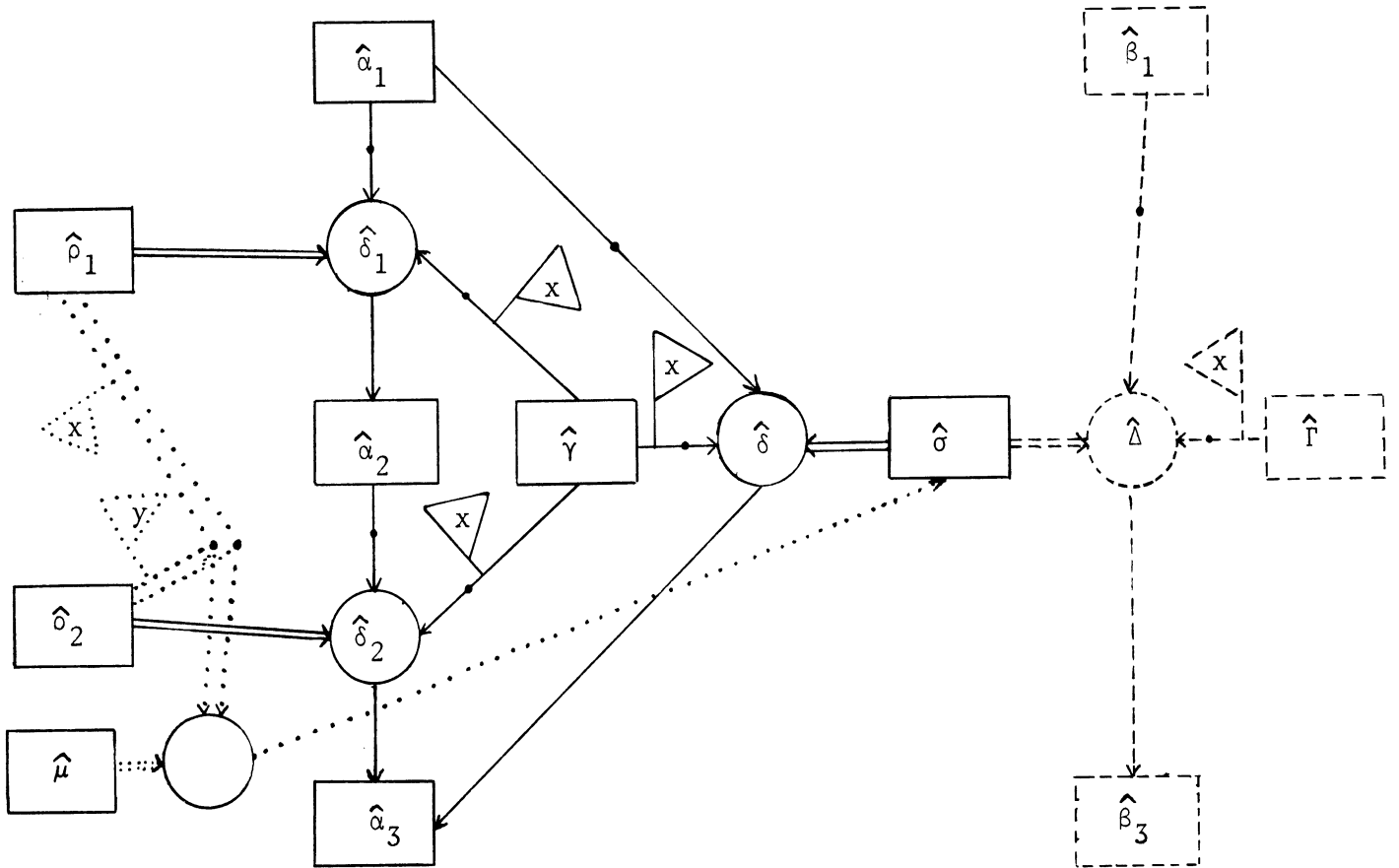


Figure 13.

The following hold:

$$\begin{aligned}
 \overline{\gamma} &= \xi \\
 \overline{\alpha_1} &= \phi(\xi) \\
 \overline{\alpha_2} &= \psi(\xi) \\
 \overline{\alpha_3} &= \theta(\xi) \\
 \overline{\rho_1} &= T(\phi(x)) \supset T(\psi(x)) \\
 \overline{\rho_2} &= T(\psi(x)) \supset T(\theta(x)) \\
 \overline{\sigma} &= T(\phi(x)) \supset T(\theta(x)) \\
 \overline{\mu} &= (T(ad(x)) \supset (add(x) \wedge T(add(x)) \supset (add(y))) \supset T(ad(x)) \supset (add(y))) \\
 \overline{\Gamma} &= \zeta \\
 \overline{\beta_1} &= \phi(\zeta) \\
 \overline{\beta_3} &= \theta(\zeta)
 \end{aligned}$$

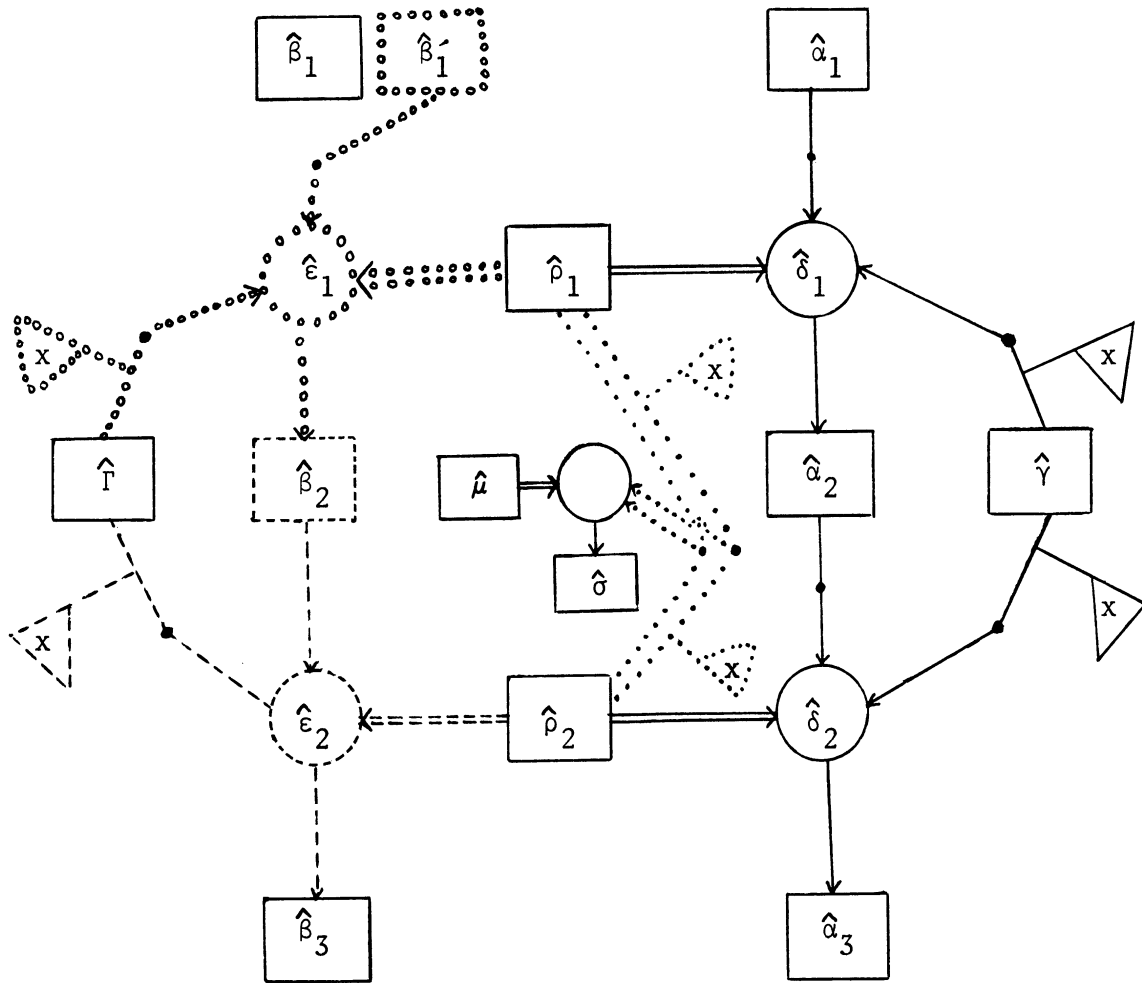


Figure 14.

The following hold:

$$\begin{aligned} \overline{\Gamma} &= \zeta \\ \overline{\beta_1} &= \phi(\zeta) = \overline{\beta_1'} \\ \overline{\beta_2} &= \psi(\zeta) \\ \overline{\beta_3} &= \theta(\zeta) \end{aligned}$$

The evaluation proceeds through the following stages (assume $\hat{\beta}_1$ and $\hat{\beta}_3$ are H-tagged; nets to be described by bug values are indicated periodically; the bug values in (1), (2), and (3) describe Figure 13 -including broken line and dotted line portions):

- (1) $Refineproof(\beta_3, \kappa)$
- (2) $Expandheuristic(\Delta, \sigma, \kappa)$
- (3) $Refinebyexample(\Delta, \delta, \sigma, \kappa)$

Now the effector makes erasures to get Figure 14, solid portion. Bug values below describe the solid and dotted (....) line portions of Figure 14.

- (4) $Prove(\beta_3, \alpha_3, \{(\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma)\}, \sigma, \kappa)$. (The third argument is a sequence, but we abbreviate it as a set so that we need write repeated members only once.)

- (5) $Parametergenerate(\beta_3, \delta_2, \{(\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma)\}, \sigma, \kappa)$
(giving $\omega := (x, \Gamma)$ in the *parametergenerate* program; see definition of *parametergenerate*.) (Assume: $result(\overline{\rho_2}, (x, \Gamma), \kappa) = \overline{\beta_3}$ holds.)

- (6) $Parametercheckgen(\rho_2, (x, \Gamma), \beta_3, \{(\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma)\}, \kappa, \delta_2, (x, \gamma), (\alpha_2), \sigma)$

Now the effector constructs the broken line (----) part of Figure 14.

Bug values below describe the solid, dotted and broken line portions of Figure 14. (*requiredantecedents* $(\rho_2, (x, \Gamma), \kappa) = \psi(\zeta)$ holds)

- (7) $Completederiv(\epsilon_2, \delta_2, \{(\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma), (\delta_2, \epsilon_2), (\alpha_2, \beta_2)\}, \sigma, \downarrow \kappa)$

(define $\Lambda =: \{(\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma), (\delta_2, \epsilon_2), (\alpha_2, \beta_2)\}$)

- (8) $Prove(\beta_2, \alpha_2, \Lambda, \sigma, \downarrow \kappa)$

(9) *Parametergenerate* ($\beta_2, \delta_1, \Lambda, \sigma, \uparrow \kappa$) (giving $\omega =: (x, \Gamma)$
in the *parametergenerate* program.)

(10) *Parametercheckgen* ($\rho_1, (x, \Gamma), \beta_2, \Lambda, \uparrow \kappa, \delta_1, (x, \gamma), (\alpha_1), \sigma$)

The effector now instructs the circled line (.....) part of Figure 14.

Bug values below describe all portions of Figure 14.

(define $\Omega =: \{ (\alpha_3, \beta_3), (\alpha_1, \beta_1), (\gamma, \Gamma), (\delta_2, \epsilon_2),$
 $(\alpha_2, \beta_2), (\delta_1, \epsilon_1), (\alpha_1, \beta_1') \}$)

(11) *Completederiv* ($\epsilon_1, \delta_1, \Omega, \sigma, \uparrow \kappa$)

(12) *Prove* ($\beta_1', \alpha_1, \Omega, \sigma, \uparrow \kappa$)

This time the effector executes *join*(β_1', β_1) to combine two nodes of Figure 14. Since $\hat{\beta}_1$ has an H-tag, so does the joined node and, defining $\phi := \text{operate}(\text{join}(\beta_1', \beta_1))$, *prove* returns the constant naming $\overline{\phi(\beta_1) * \phi(\Omega)}$.

Undoing the recursions, all the *H-tagged*($\alpha(\zeta)$) occurrences hold because *completederiv* H-tags $\hat{\beta}_2$ and $\hat{\beta}_3$ in turn. *Prove* finally gives an answer of form (β_3, Ξ) (for some sequence Ξ) where an H-tag is on $\hat{\beta}_3$ and the net constructions we have mentioned have been made. This is returned to *refinebyexample* which restores the erased connections and calls *refineproof*($\beta_3, \uparrow \kappa$)

where β_3 is the new β_3 with the new constructions. The original problem (refining the $\hat{\sigma}$ step) has been broken into two subproblems (refining the $\hat{\rho}_2$ step and refining the $\hat{\rho}_1$ step).

3.6.5 Less Trivial Situations

In the example, we had an entire simple model laid out before us to mimic. When this is not the case, or when the right model is buried among

incorrect models, the task is harder and the chances for error greater. Such cases require random searches through the net to find the required formula. Such search algorithms can be sophisticated or simple. We have only indicated where they are used. We have used them in sequential rather than diagonal manner, since we are not trying to set up the ultimate in search algorithms here but only indicate the principles by which they operate. The random search algorithms are, *Rcheck*, *overallcheck*, and *randomprove*. Each searches for a different kind of formula. Details are discussed below.

3.6.6 *parameter tree generate*

Even when an entire model is available, and no random net searches are required, the situation can be rather complicated. Good heuristics, however, give us simple situations. In the previous example we had to break up a single step heuristic derivation into a two-step one. Sometimes one must break a single step into three steps, but a good heuristic would be in the net in such a way that this process is done in two *refineproof* stages, first breaking the single step into two, and then breaking one of these into two more.

In general, the complicated entire model situations are handled just like the example. There is one situation, however, which is common, but did not arise in the example. When the effector arrived at *parameter generate* it always found that $ad(v) \in \overline{proj_1(\ell)}$ held, for the current values of dummy variables v and ℓ . One can't always count on this. Suppose the model were as in Figure 15. Here $ad(v)$ names γ . γ is not in $\overline{proj_1(\ell)}$, but it is a derivative of a member of $\overline{proj_1(\ell)}$, the derivation being via rule $\overline{\rho}$. We call such a rule a parameter rule because it generates a parameter value.

Such rules are easy to recognize since they are all of form

$(T(x_1) \wedge T(x_2) \wedge \dots \wedge T(x_n)) \supset T(\Psi(x_1, x_2, \dots, x_n))$ for some individual function Ψ . Hence it is trivial to follow a model back through one of these rules and mimic it in the new proof. This is the job of *parameter-treegenerate*. *etacheck* then tries to use the generated parameter value for the job it is supposed to do. Of course, our definitions are simplified since we are assuming the rule contains only one free variable, only one parameter.

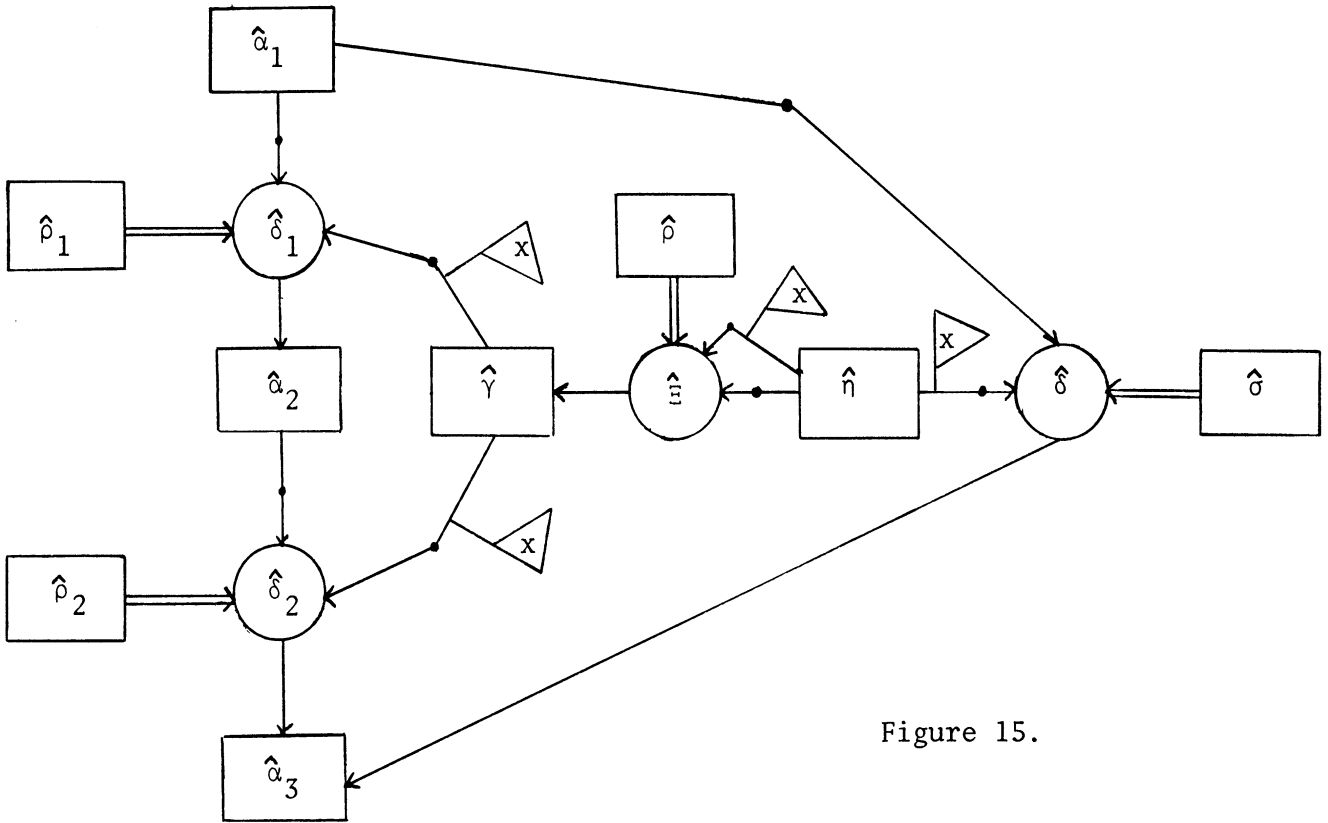


Figure 15.

The following hold:

- $\overline{\eta} = \xi$
- $\overline{\gamma} = \phi(\xi)$
- $\overline{\alpha_1} = \phi(\phi(\xi))$
- $\overline{\alpha_2} = \psi(\phi(\xi))$
- $\overline{\alpha_3} = \theta(\phi(\xi))$
- $\overline{\rho} = T(x) \supset T(\phi(x))$
- $\overline{\rho_1} = T(\phi(x)) \supset T(\psi(x))$
- $\overline{\rho_2} = T(\psi(x)) \supset T(\theta(x))$
- $\overline{\sigma} = T(\phi(\phi(x))) \supset T(\theta(\phi(x)))$

It is important to distinguish parameter rules from other rules since they perform quite a different function and are really only individual function type expressions written in rule form.

We make the simplifying assumption that it will never be necessary to search through a cascaded series of parameter rules to get to the member of $\overline{proj_1(\ell)}$ we want. In general, this makes a lot of sense because we can automatically arrange our construction rules to collapse down a cascaded series into one rule, the process being trivial, recursive, and reversible. The result merely shows composed individual functions to the right of the implication sign. I have not taken the trouble to arrange for the collapsing, but I have arranged the random search functions (which generate these rules) in such a way that cascades do not develop. The parameter rules are treated like individual functions, not like real rules.

3.6.7 Suppose the Model Fails at Some Point

Suppose the effector is looking for a formula to fit in the proof, and none of the corresponding parts of the model proof lead it to members of $\overline{proj_1(\ell)}$ (ℓ having the current value for the dummy variable ℓ .) The model has failed at this point. Before the effector rejects the whole model, it might be a good idea for it to take a look at nearby formulae to see if they are what it is looking for. When no model exists, the whole proof attempt must consist of this sort of search. The situation is not as bad as it seems. In Figure 13, for example, even if none of the solid lines existed (i.e., if there was no model) a search of the net near $\hat{\sigma}$ would quickly bring the effector to the correct rules of inference.

The random search, then, begins at an appropriate node or nodes and searches nearby nodes first. It should search first those nearby nodes with high values on value-type flags. (Each random search algorithm looks at a

different set of flags.) The algorithm looks at a number parameter κ , which tells it, in effect, how thoroughly to search. By the rewarding and punishing routine, the search avoids repeating a trial too often. (We can fix things so it never repeats a trial if we wish.) The algorithm carries along the ℓ sequence. The algorithm will be constructing new nodes and hooking them onto old ones. If these old ones are in $\overline{proj_2(\ell)}$ then this is good to know because at that point the effector can stop its random search and go back to the model. (I have indicated this in the algorithms by a recursion back to *prove* and hence to *checklistprove*, but I have indicated it only for antecedents.)

Theoretically, given a high enough κ , these random searches could search the whole net. But the net could easily have become split in two parts by erasure of certain nodes. We must keep track of both halves, not only to prevent one part from being garbage collected, but to allow the random search algorithms to search both halves.

I have indicated three random-search algorithms.

Type of Formula Sought	Algorithm Name	Selecting Function Used
normal rule	<i>randomprove</i>	<i>bestnetrule</i>
parameter rule	<i>overallcheck</i>	<i>bestnetparameterrule</i>
parameter value	<i>Rcheck</i>	<i>bestnetparametervalue</i>

The first is more general and incorporates the other two. It can be used when there is no model at all. The selecting functions are like *bestlist*, but instead of selecting nodes from a sequence they select them from the whole net, weighting them as we discussed above. The first argument of a selecting function names a bug value (or list of bug values in *bestnetparameter-value*) designating a node (or nodes) around which the effector will concentrate its search. The second argument names the number parameter.

As we mentioned, *bestnetparameterrule* is rigged to discourage cascading. If it doesn't immediately find what it wants, a nearby parameter rule with high value, it constructs one. Remember that the rule is just an individual function. Usually the required individual function will be buried in a nearby node (e.g., see Figure 8 where the function ϕ , for $\overline{\rho}$, is buried in $\overline{\rho}$). *Bestnetparameterrule* actually searches through selected rules for the proper individual function and constructs the required parameter rule via the obvious meta rule. It returns a bug value designating the constructed node. Since the effector will never want to T-tag the rule, this odd generation causes no problems. If this sort of search fails, *bestnetparameterrule* searches simultaneously (and diagonally) for several individual functions (the search for each being like the above search) and composes them to create a variable rule. Unless κ is rather high, this search will be quickly given up.

3.7 General Considerations

A more detailed discussion of the algorithms here presented would be pointless since their definitions are only a simplified outline of one possible approach to effector interpretation of heuristics. Much more sophisticated algorithms are possible. The more sophisticated the effector algorithms, the simpler the heuristics and memory net structure need be. Simple algorithms imply complicated models with many intermediate stages in proof-refining between the original heuristic proof outline and the final rule of inference proof. Since we are interested in adaptation of the net, we presumably would like rather simple effector algorithms (i.e., we don't want to program too many techniques into the machine; we want the machine to discover them for itself), but not as simple as the algorithms presented here. The *prove* algorithm presented here does exhibit an important property shared with its more complicated relatives. We described this property in Section

1.6.3.2. In terms of our notation here we can summarize the property by saying that *prove* never looks at T-tags, only at H-tags.

3.8 Conclusion: Discussion of Adaptation. Notice that by the reward and punishment scheme not only is the effector able to make random searches without always repeating itself, but when it has completed a proof (if we write the algorithms correctly) it will have rewarded (or punished if it was unsuccessful) just those nodes we would want it to try (or avoid) next time; those strategies which work well get higher reward and are more likely to be tried in the future. The exact reward for each of the rewarded nodes can be modified by a multiplier (the same multiplier for each node) set by the user on the basis of how good a proof was produced. Other effector algorithms can then help further distribute the reward. The idea, of course, is to erase, from time to time, very poorly rewarded nodes. Thus the net adapts slightly with each problem it solves or fails to solve.

There is a special effector algorithm which, after completion of a "to prove" problem, makes random applications of rules in the net to formulae in the net, thus generating new formula nodes. We shall call this algorithm the random generation algorithm. This algorithm makes a random choice of a formula node containing a rule (weighting as more likely to be chosen those nodes which have accumulated more reward). It then makes a random choice of formula nodes to apply the rule to (weighting as more likely to be chosen those nodes which, have accumulated more reward and are closer to the rule chosen). The algorithm attempts to apply the chosen rule to the chosen formulae. If the application is successful, the algorithm adds to the net a formula node containing the resulting formula and a derivation node whose connections show how the new formula was derived.

Thus the random generation algorithm is very simple. Nevertheless it allows the machine great flexibility in adaptation. By its operation, formulae are added to the net in areas which the machine itself has determined (via its reward distribution algorithms) are important areas. The formula nodes created are not for use in the solution of a current problem, but are rather for use in the solution of future problems which are similar to those problems in whose solution the formula nodes chosen by the algorithm participated.

One way in which these new formulae might be used in future problems would be as rules. If the new formula generated is in the form of a rule, it can easily be employed as a rule in future problems. If such a new formula was generated from theorems by a rule of inference, it can be used as a new "derived" rule of inference. If not, and this is the more common case, it can still be used as a heuristic.

In Section 1.7 we discussed the generation of new heuristics by the machine. It is usually through the random generation algorithm that such a generation

takes place. Figure 8 diagrammed the result of such a generation. We can now see that the derivation diagrammed in Figure 8 is no different from any step in a proof outline. It simply happens that the formula generated is in the form of a rule. We see now that the rule generated (Heuristic 1 in Figure 8) may be either closely related or quite unrelated to the rules (Rule 1 in Figure 8) and to other formulae (the * in Figure 8) from which it was generated; the degree and nature of the relationship is determined by the rule which was used as a rule in the derivation. (The node containing this rule is not shown in Figure 8. It is the node from which emanates the lower of the two thick arrows in Figure 8.) We can call this last rule a meta rule since it was used to generate a rule. A rule which is used to generate a meta rule can be called a meta meta rule. Thus we can think of a rule as being used at a particular level: the object level, the meta level, the meta meta level, or a still higher level.

Now a single rule may be used successively at any number of different levels. Thus, it is not the rule which is at a particular level, but only a particular use of it. Of course, there will be some rules useful at a particular level and useless at other levels. This specialization is essential to proper adaptation. Efficient generation of new rules at any particular level depends on the presence of rules useful at the next higher level. As a corpus of rules is developed, useful at a particular level ℓ , the generation of rules at the next lower level (level $\ell - 1$) becomes efficient. Until this corpus is developed (through reward of rules at level ℓ which produce rules rewarded at level $\ell - 1$) the generation of rules at level $\ell - 1$ is inefficient. Note that though the generation of rules at level $\ell - 1$ is inefficient to begin with, it still takes place, since any rule in the net can be used at level ℓ . (In fact, we suspect that if level ℓ is a very high level, any rule useful at level $\ell - 1$ is useful at level ℓ .) Thus, the net is never in a position of having no rules to use at a given level, since

any rule can be used at any level. It is only after a good deal of adaptation has taken place that rules are developed which are specialized for use at a particular higher level. Rules specialized for use at the lower levels will appear first (at the object level almost immediately) and only later will rules specialized for use at the higher levels appear. Thus a hierarchy of rules is gradually built up.

(We have been talking as though, given a particular use of a rule, it were always possible to tell at what level the use occurred. This is, of course, not always the case, especially before the hierarchy is built up. Suppose Rule A is used to generate Rule B. Then suppose Rule B is used once at the meta level and once at the meta meta level. At what level was Rule A used when it was used to generate Rule B? However, it is useful to think of the use of a rule as if it occurred at a particular level.)

Now a generated rule may be either a heuristic or a rule of inference. If it was a heuristic, its "proof" must have been only a proof outline (it can never be fully refined). Even if the generated rule was a rule of inference (and hence a theorem) its "proof" might not be fully refined. (This was almost certainly the case when the rule was first generated.) In such a case, the rule is a rule of inference, but, since its node does not have a T-tag, the effector does not know that it is anything more than a heuristic. If it is a useful heuristic, however, the effector will spend time trying to refine its proof. This is because the rule will accumulate reward and thus there will be many attempts to use it. At each attempt, the effector makes another try at refining the rule's proof outline. (For the object level, see the command to calculate ρ in the *refineproof* routine, Section 4.11.2. For the meta level and higher levels we need other effector algorithms.)

By allowing any given rule to be used on any one level or several different levels, we can start the machine with an especially simple memory--with a single corpus of rules to be used on all levels--and then allow the corpus of rules to specialize. (Alternatively, we could have restricted each rule to a given level, beginning each level with the same original corpus of rules. This plan, however, does not allow a new rule useful at one level to be tried out at another level.)

Our general procedure for generating new rules is what permits the development of a hierarchy of rules. That rules generated may be not only heuristics, but even rules of inference, depends on the procedure (discussed in Section 2.2.1.2) for adding new "derived" rules of inference to the axiomatic system. (The procedure for generating heuristics is merely a generalization of the procedure for generating "derived" rules of inference.) This procedure in turn depends on both the procedure for generating algorithmic names of functions and on the procedure for using Rules 18 and 19 to shift a theorem from one level to another in our language. All these procedures were explained in Section 2 and it was shown in Section 2.2.1, that addition of these procedures to our axiomatic system does not destroy consistency.

By showing, in Section 2, the existence of a consistent self-describing system which incorporates these procedures, we have shown that there exists, an axiomatic system which may be used as a basis for the sorts of adaptive theorem proving machines discussed in Sections 1 and 3.

The complete specification of one such adaptive theorem prover is not possible until various reward plans have been investigated and compared. We are only beginning this investigation and it promises to be a long one. Our initial work will be on the so-called "reproductive plans" which Dr. Holland has investigated (see description in [Holland, 1969]).

3.9 Postscript: Other Object Theories

Formally, the machine proves theorems in a particular axiomatic system. We can cause it to prove theorems in another system simply by adding that system's axioms and rules of inference to the memory net in the following way. We define a predicate *Sys* in the same way we defined *T* except that the definition uses the axioms and rules of inference of the new system instead of the old system. Then *Sys* is true on theorems in the new system. If we want the machine to prove a theorem α of the new system, we simply ask the machine to prove $Sys(\alpha)$.

The old system is present, then, "overseeing" the new system and operating at the meta level to generate new rules of inference and heuristics for the new system.

The new system, then, is "subordinate" to the old. There is no counterpart to Rule 19 for *Sys* so the new system stays at the object level, as it were. Rules of inference for the new system are theorems of the old system, not theorems of the new system.

By the above method we can have the machine prove theorems in trigonometric identities, group theory, or propositional calculus.

In the case of propositional calculus, the new system is a subsystem of the old system, so for any α , $Sys(\alpha) \supset T(\alpha)$ holds. As long as all the special propositional calculus rules of inference require the generated theorem to be a formula of propositional calculus (a decidable question and thus easy to add as a condition, see below), we can have the machine use *T* instead of *Sys* throughout.

Let us consider the Newell, Shaw, Simon [Newell, Shaw, and Simon, 1961] formulation for propositional calculus. (Ignore for now their so-called abstract operators.) They formulate problems in terms of transforming a

propositional calculus formula α into β by means of (reversible) legal transformations. These transformations, or operators, we may write as individual functions op_1, op_2, \dots, op_n . Since our machine works with theorems, we shall attempt to transform the propositional calculus theorem $\alpha \equiv \beta$ into theorem $\alpha \equiv \alpha$ by means of applying the op_j 's to the right hand side. To decide what operation to use, Newell, Shaw, and Simon have a set of difference tests which we can regard as a set of binary predicates D_1, D_2, \dots, D_m .

Let us make the following definitions.

$$D_0(x,y) =: \sim (D_1(x,y) \vee \dots \vee D_n(x,y))$$

$prop(\alpha)$ holds iff α names a well formed formula of propositional calculus.

$$diff(x,y) =: [prop(x) \wedge \sim D_0(x,y) \rightarrow x \left(\equiv \right) y] ;$$

$$atom(x) \rightarrow \emptyset ;$$

$$diff(a(x), a(y)) \neq \emptyset \rightarrow diff(a(x), a(y)) ;$$

$$\oplus \rightarrow diff(d(x), d(y))]$$

There are many ways of embedding a fixed scheme such as Newell, Shaw, and Simon's in our system. One is to code it all into one massive rule of inference. This would only be interesting if the machine then dissected it into a set of smaller rules. Let's begin with such a set of smaller rules.

We shall ignore, in this illustration, their three rules which use two antecedents. Since the second antecedent is found by random search, the models for these rules are essentially the same as for other rules. (Eg., see Fig. 5, Sec. 1.) (The $\left(\equiv \right)$ then become $\left(\supset \right)$ everywhere.) (If these three rules are also to operate on subexpressions, further modification is necessary.)

The basic rules of inference are, for each j such that $1 \leq j \leq n$:

$$\rho_j \quad (prop(x) \wedge prop(y) \wedge T(x \stackrel{\ominus}{\equiv} op_j(y))) \supset T(x \stackrel{\ominus}{\equiv} y). \quad \text{In addition we need}$$

$$\varepsilon \quad (prop(x) \wedge x = y) \supset T(x \stackrel{\ominus}{\equiv} y).$$

Associated heuristics will be

$$\nu \quad \alpha(z) = \stackrel{\ominus}{\equiv} \supset T(z) \quad (\text{like } \oplus \supset T(x) \text{ in the old system})$$

$$\omega \quad \oplus \supset T(x \stackrel{\ominus}{\equiv} y)$$

$$\pi_i \quad D_i(x, y) \supset T(x \stackrel{\ominus}{\equiv} y) \quad (\text{one for each } i \text{ such that } 0 \leq i \leq m)$$

$$\tau_{ij} \quad (D_i(x, y) \wedge \sim D_i(x, op_j(y))) \supset T(x \stackrel{\ominus}{\equiv} y) \quad (\text{one for each } i, j, \text{ such}$$

that $1 \leq i \leq m$ and $1 \leq j \leq n$)

When $D_0(\alpha, \beta)$ holds, Newell, Shaw, and Simon stop trying to transform the β into α directly by the op_j 's, and begin trying to similarly transform its sub-expressions. We can generate such subgoals by properly employing the following rule of inference

$$\sigma \quad (prop(x) \wedge prop(y) \wedge T(diff(x, y)) \wedge T(S(add(diff(x, y)), ad(diff(x, y)), x) \stackrel{\ominus}{\equiv} S(add(diff(x, y)), ad(diff(x, y)), y))) \supset T(x \stackrel{\ominus}{\equiv} y)$$

These rules (and no axioms) are sufficient for our machine to essentially simulate the Newell, Shaw, and Simon techniques. The simulation will be most efficient if the net contains various "models" of the sort we discussed above. The models in Figures NSS1-NSS6 are sufficient to force the machine to follow a probabilistic Newell, Shaw, and Simon algorithm. (In these figures, the symbol in a node indicates the node's contents, not its name.) Adaptation causes changes in the value-type flags and hence changes in strengths of connection between the D_i 's and op_j 's. (Of course, we can prevent such adaptation by simply returning the net to its original state after each problem.) In these models the rules frequently have two free variables so the simplified *refineproof* algorithm of the last example won't work here. There is no essential difference, however, between the model-following technique of this example and that of the last.

One model:

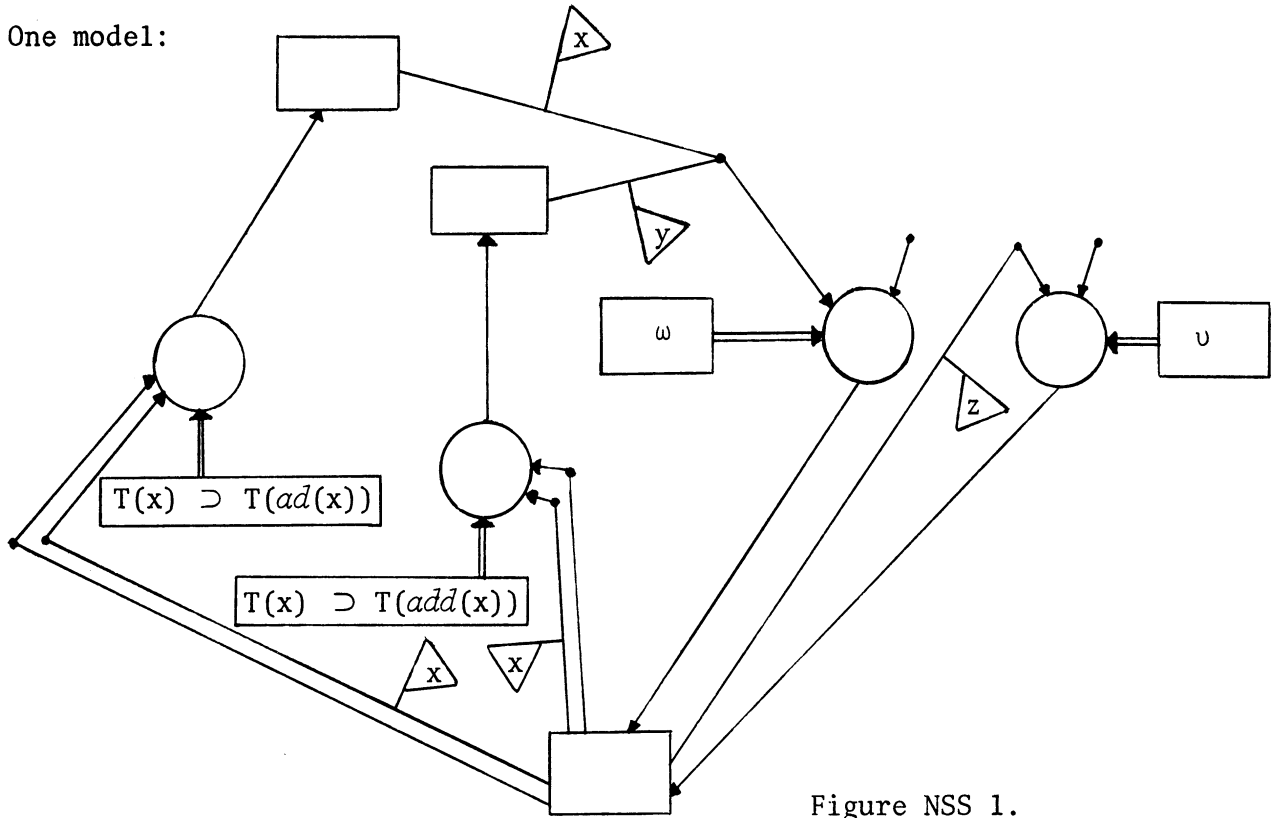


Figure NSS 1.

A model for each π_i , even for $i = 0$:

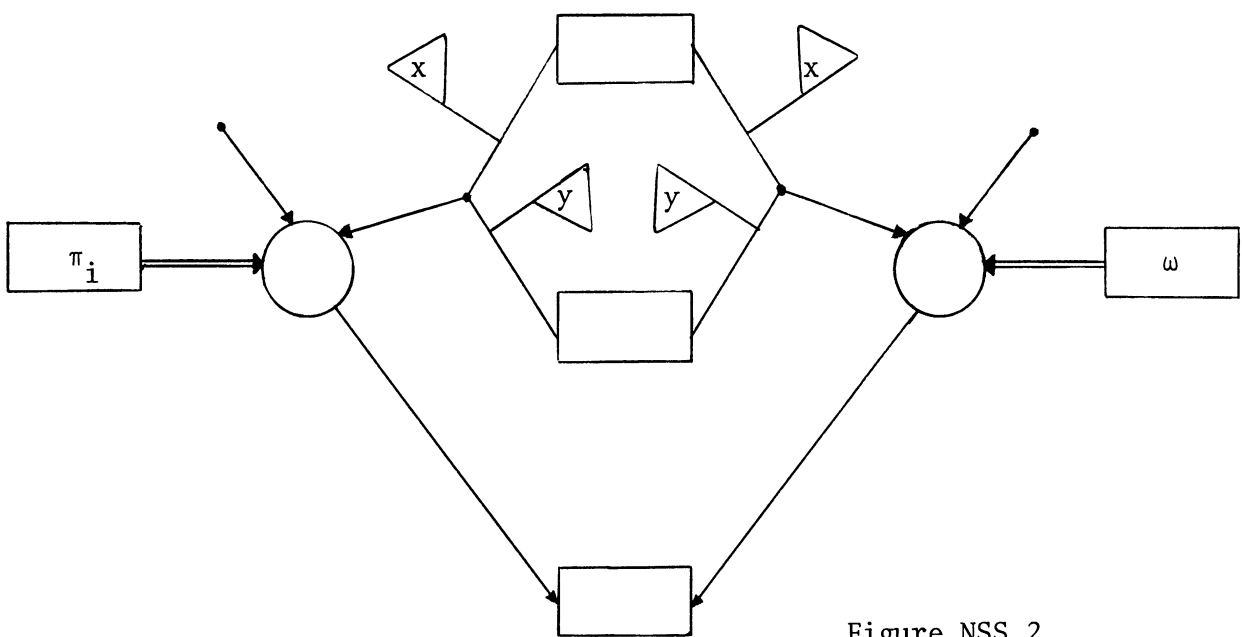


Figure NSS 2.

A model for each τ_{ij} ($i \neq 0$):

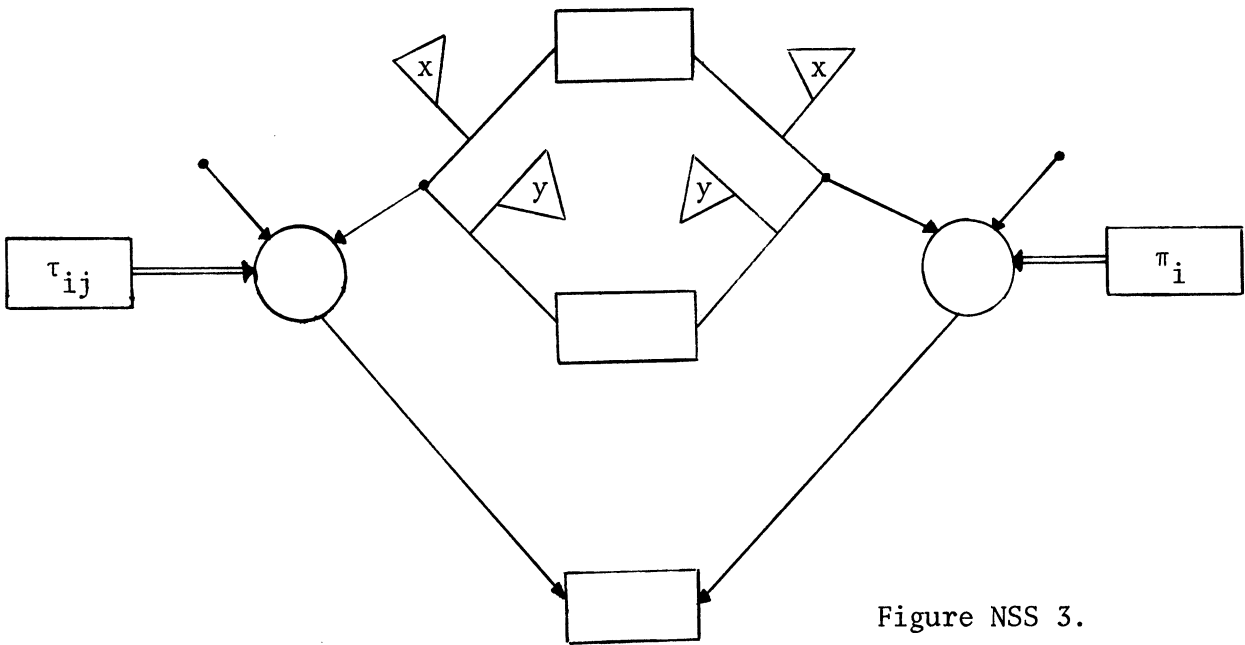


Figure NSS 3.

A model for each τ_{ij} ($i \neq 0$):

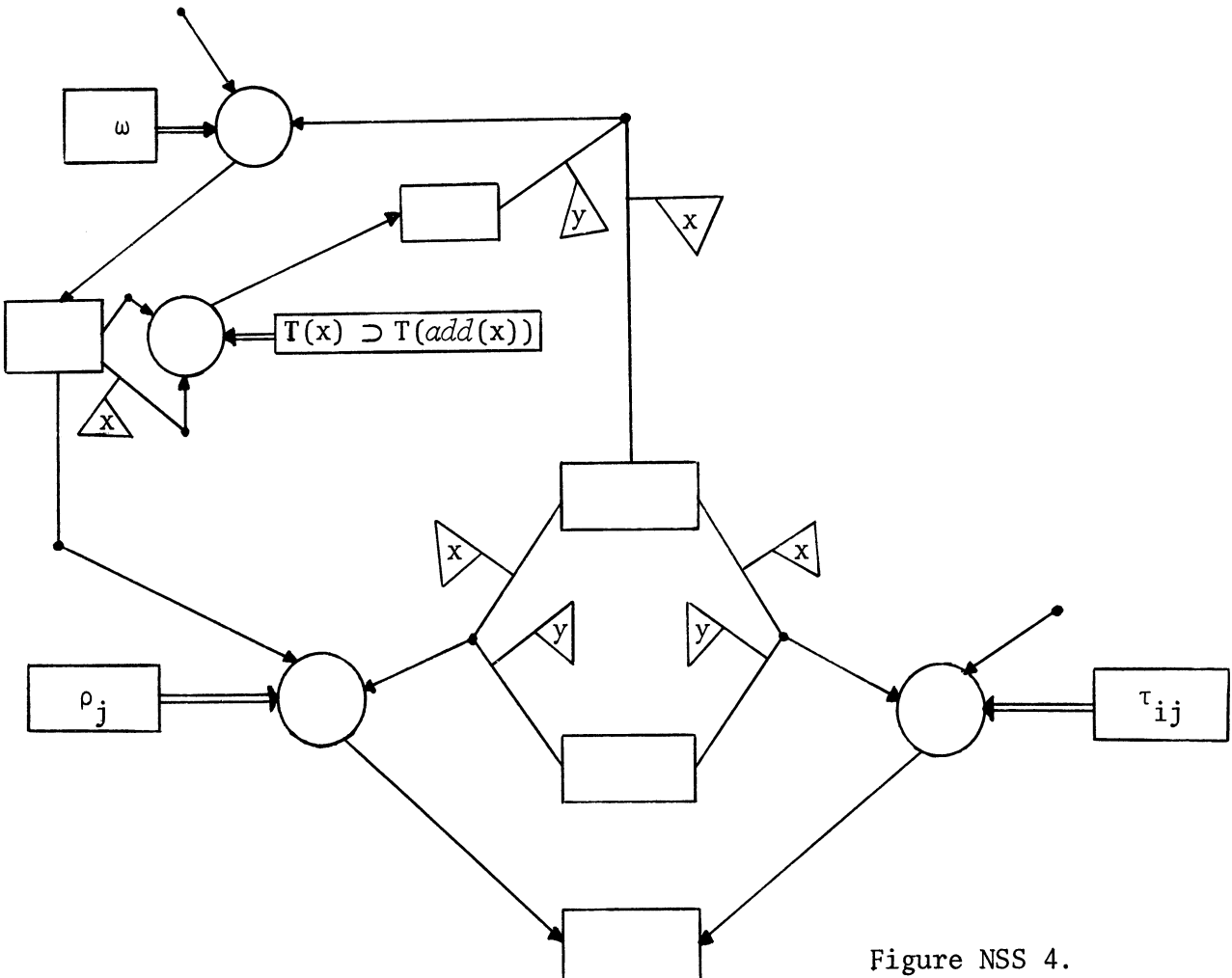


Figure NSS 4.

There are two models for π_0 . Here is one:

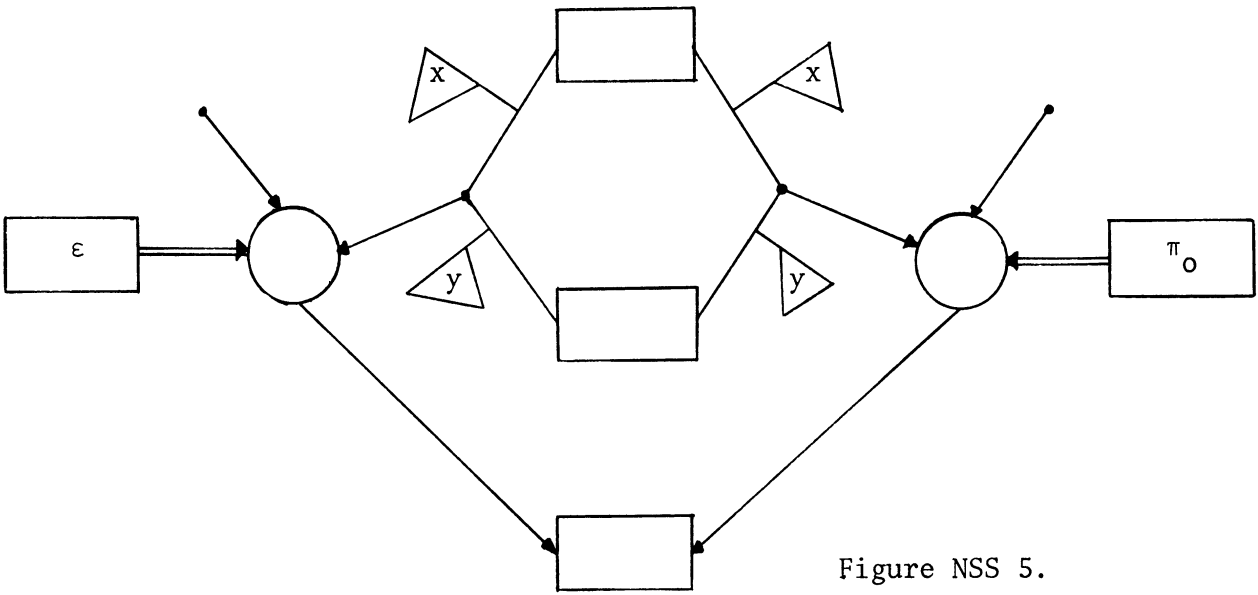


Figure NSS 5.

Here is the second model for π_0 :

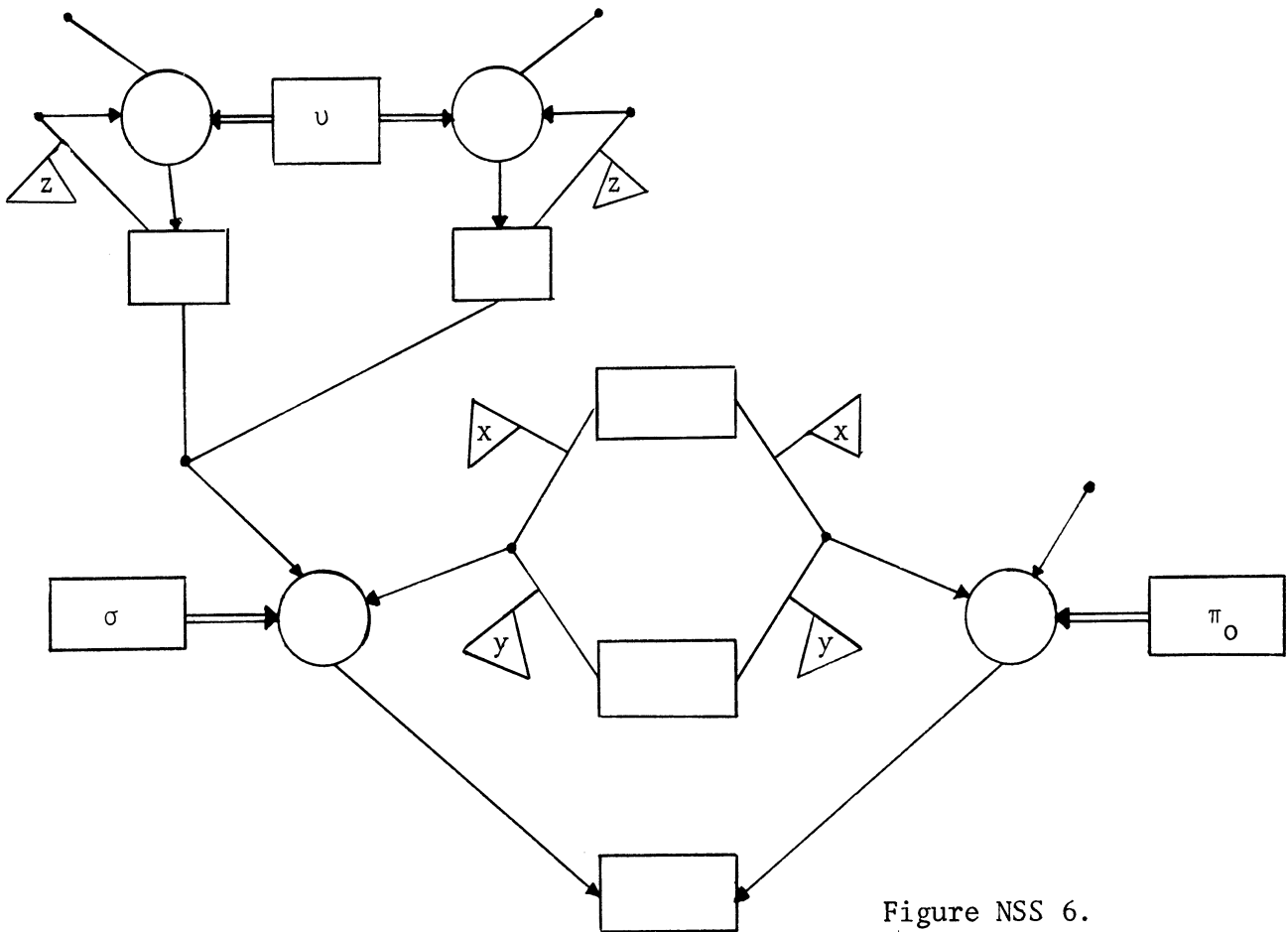


Figure NSS 6.

Thus a simple net may be constructed which simulates the Newell, Shaw, and Simon system, and this construction does not need most of the axioms and rules of the old system. If these are added, however, then we have the possibility of adding new rules of inference and heuristics. Newell, Shaw, and Simon point out that if for many α 's, $\Pi(\alpha, op_j(\alpha))$ holds, then Π is a good candidate for a new D_i . If we have our old system in the net, the machine can prove the statements of form $\Pi(x, op_j(x))$. The following meta rule will convert such a statement into a rule like the τ_{ij} 's:

$$(operator(y) \wedge T(\langle x, x, (y, x) \rangle)) \supset T(\langle \langle x, x, y \rangle \wedge \sim \langle x, x, (y, y) \rangle \rangle \supset T(x, \langle \langle y \rangle \rangle))$$

where we define

$$operator(y) =: T(\langle T(x, \langle \langle y \rangle \rangle) \supset T(x, \langle \langle y \rangle \rangle) \rangle)$$

(i.e. y doesn't have to be one of the original operators; it can be a new one we've generated).

For practice one would want $\Pi(x, op_j(x))$ to be a theorem for more than one j . It need not, however, hold for all x , but perhaps only for a subset of all x . If ϕ defines the subset then instead of $\Pi(x, op_j(x))$ we would perhaps only require that

$$\phi(x) \supset (\Pi(x, op_j(x)) \wedge \Pi(x, op_k(x))) \quad \text{be a theorem.}$$

Much less ambitious than this would be simply adding meta rules to combine the D_i 's and op_j 's in various ways to derive new D_i 's and op_j 's. Also, we can introduce abstract operators. These can be simply introduced as intermediate rules between the Π_i 's and ρ_j 's. Some care must be taken, however, in setting up the models. The power of the abstract operators is that one can't H-tag the formula one is trying to prove until one has a complete derivation via abstract operators. Hence the machine must not call in \cup or ω in the models because they give a false H-tag. (This is a general problem, not limited to

the Newell, Shaw, and Simon net.) This can be fixed by more complicated modeling and by low values on nodes containing \cup and ω so *bestnetrule* won't find them. This is an example of the sort of difficulty that could be solved by diagonalization of *refineproof* so that instead of always refining the last illegitimate step of a proof outline, it refined the most illegitimate step, as measured by value-type flags on the heuristics.

4. TABLES

4.1 Table 1. Alphabet

p, q, r, s, p, \dots	(propositional variables)
$x, y, z, u, v, w, x_i, \dots$	(individual variables)
f, g, h, f_i, \dots	(individual function variable bases)
P, Q, R, P_i, \dots	(predicate function variable bases)
\oplus, \otimes	
nil	
$\supset, =, pv, Iv, Ifvb, Pfvb$	
$*, newpv, newiv, newifvb, newpfvb$	
cond, pcond	
qu	
$\forall, \exists, \lambda$	(listbinders)
$\exists!, i, label$	

4.2 Table 2. Basic Recursive Functions

The basic function expressions which are complete recursing by virtue of special evaluation procedures. (Any machine employing our system would have these procedures stored in it in a manner similar to the way LISP SUBR's are stored.)

<u>function expressions</u>	<u>type</u>
\supset	(Pfvb,Pv,Pv)
=	(Pfvb,Iv,Iv)
Pv	(Pfvb,Iv)
Iv	(Pfvb,Iv)
Pfvb	(Pfvb,Iv)
Ifvb	(Pfvb,Iv)
*	(Ifvb,Iv,Iv)
α	(Ifvb,Iv)
d	(Ifvb,Iv)
} "defined" in Table 3	
newpv	(Ifvb,Iv)
newiv	(Ifvb,Iv)
newpfvb	(Ifvb,Iv)
newifvb	(Ifvb,Iv)

Meanings of the Predicate expressions:

<u>formula</u>	<u>holds if and only if</u>
$\alpha \supset \beta$	β holds or α does not hold
$\alpha = \beta$	α and β name the same S-expression
Pv(α)	α names a propositional variable
Iv(α)	α names an individual variable
Pfvb(α)	α names a Predicate function variable base
Ifvb(α)	α names an individual function variable base

Meanings of the individual function expressions:

<u>term</u>	<u>names</u>
$\alpha * \beta$	the cons of $\overline{\alpha}$ and $\overline{\beta}$
$a(\alpha)$	the car of $\overline{\alpha}$
$d(\alpha)$	the cdr of $\overline{\alpha}$
$newpv(\alpha)$	the first prepositional variable of index > those in $\overline{\alpha}$
$newiv(\alpha)$	the first individual variable of index > those in $\overline{\alpha}$
$newpfb(\alpha)$	the first predicate function variable base of index > those in $\overline{\alpha}$
$newifvb(\alpha)$	the first individual function variable base of index > those in $\overline{\alpha}$

4.3 Table 3. Defined Complete, Recursive Functions of a General Nature

Complete recursing function expressions:

$$\sim p =: p \supset \textcircled{\emptyset}$$

$$p \vee q =: \sim p \supset q$$

$$p \wedge q =: \sim(p \supset \sim q)$$

$$p \equiv q =: (p \supset q) \wedge (q \supset p)$$

$$listbinder(x) =: x = \textcircled{\lambda} \vee x = \textcircled{\iota} \vee x = \textcircled{\mathbb{H}} \vee x = \textcircled{\forall} \vee x = \textcircled{\mathbb{H}!}$$

$$predatom(x) =: x = \textcircled{Pv} \vee x = \textcircled{Iv} \vee x = \textcircled{Ifvb} \vee x = \textcircled{Pfvb}$$

$$newfatom(x) =: x = \textcircled{newpv} \vee x = \textcircled{newiv} \vee x = \textcircled{newpfvb} \vee x = \textcircled{newifvb}$$

$$Pfatom(x) =: Predatom(x) \vee x = \textcircled{\supset} \vee x = \textcircled{=}$$

$$Ifatom(x) =: newfatom(x) \vee x = \textcircled{*}$$

$$nonvaratom(x) =: x = \textcircled{\oplus} \vee x = \textcircled{\ominus} \vee x = \textcircled{cond} \vee x = \textcircled{pcond} \vee x = \textcircled{label}$$

$$\vee listbinder(x) \vee Pfatom(x) \vee Ifatom(x) \vee x = \textcircled{qu} \vee x = \emptyset$$

$$atom(x) =: Pv(x) \vee Iv(x) \vee Pfvb(x) \vee Ifvb(x) \vee nonvaratom(x)$$

$$a(x) =: \iota(y) ((atom(x) \wedge y = x) \vee (\mathbb{H}(z) y * z = x))$$

$$d(x) =: \iota(y) ((atom(x) \wedge y = x) \vee (\mathbb{H}(z) z * y = x))$$

$$x \neq y =: \sim x = y$$

$$last(x) =: [atom(x) \rightarrow x; \textcircled{\oplus} \rightarrow last(d(x))]$$

$$length(x) =: [atom(x) \rightarrow \emptyset; \textcircled{\oplus} \rightarrow \textcircled{p} * length(d(x))]$$

$$andlistcar(P_{Iv}, x) =: atom(x) \vee (P_{Iv}(a(x)) \wedge andlistcar(P_{Iv}, d(x)))$$

$$andlista(P_{Iv}, x) =: andlistcar(P_{Iv}, x) \wedge last(x) = \emptyset$$

$$andlistlistcar(P_{IvIv}, x, y) =: x = \emptyset \vee (\sim atom(x) \wedge \sim atom(y))$$

$$\wedge P_{IvIv}(a(x), a(y)) \wedge andlistlistcar(P_{IvIv}, d(x), d(y))$$

$$andlistlista(P_{IvIv}, x, y) =: andlistlistcar(P_{IvIv}, x, y) \wedge length(x) = length(y)$$

$$\wedge last(x) = \emptyset \wedge last(y) = \emptyset$$

$$ast(x) =: [atom(d(x)) \rightarrow a(x) ; \textcircled{\oplus} \rightarrow ast(d(x))]$$

$$\text{maplistcar}(f_{IV}, x) =: [\text{atom}(x) \rightarrow x; \text{⊕} \rightarrow f_{IV}(a(x)) * \text{maplistcar}(f_{IV}, d(x))]$$

$$\text{contains}(P, x, y) =: x = y \vee (\sim \text{atom}(y) \wedge \sim P(y) \wedge (\text{contains}(P, x, a(y)) \vee \text{contains}(P, x, d(y))))$$

$$\text{Subst}(P, x, y, z) =: [x = z \rightarrow y; \text{atom}(z) \vee P(z) \rightarrow z; \text{⊕} \rightarrow \text{maplistcar}((\lambda(u) \text{Subst}(P, x, y, u)), z)]$$

$$\text{reverseconc}(x, y) =: [\text{atom}(x) \rightarrow y; \text{⊕} \rightarrow \text{reverseconc}(d(x), a(x) * y)]$$

$$\text{orlistcar}(P_{IV}, x) =: \sim \text{andlistcar}((\lambda(x) \sim P_{IV}(x)), x)$$

$$x \in y =: \text{orlistcar}((\lambda(u) u = x), y)$$

$$\text{typep}(x) =: x = \text{Pv} \vee x = \text{Iv} \vee ((a(x) = \text{Pfvb} \vee a(x) = \text{Ifvb})) \wedge \text{andlista}(\text{typep}, d(x))$$

$$\text{typelist}(x) =: \text{andlista}(\text{typep}, x)$$

$$\text{Pfv}(x) =: \text{Pfvb}(a(x)) \wedge \text{typelist}(d(x))$$

$$\text{Ifv}(x) =: \text{Ifvb}(a(x)) \wedge \text{typelist}(d(x))$$

$$\text{variable}(x) =: \text{Pv}(x) \vee \text{Iv}(x) \vee \text{Pfv}(x) \vee \text{Ifv}(x)$$

$$\text{varlist}(x) =: \text{andlista}(\text{variable}, x)$$

$$\text{type}(x) =: [\text{Pv}(x) \rightarrow \text{Pv}; \text{Iv}(x) \rightarrow \text{Iv}; \text{Pfv}(x) \rightarrow \text{Pfvb} * d(x); \text{Ifv}(x) \rightarrow \text{Ifvb} * d(x); \text{⊕} \rightarrow \emptyset]$$

$$\text{newvart}(x, y) =: [x = \text{Pv} \rightarrow \text{newpv}(y); x = \text{Iv} \rightarrow \text{newiv}(y); \text{atom}(x) \rightarrow \emptyset; a(x) = \text{Pfvb} \rightarrow \text{newpfvb}(y) * d(x); a(x) = \text{Ifvb} \rightarrow \text{newifvb}(y) * d(x); \text{⊕} \rightarrow \emptyset]$$

$$\text{exprtype}(x) ::= [x = \textcircled{\text{⊕}} \vee x = \textcircled{\text{⊕}} \rightarrow \textcircled{\text{Pv}};$$

$$\text{predatom}(x) \vee \text{newfatom}(x) \rightarrow \textcircled{\text{Pfvb}}_{\text{Iv}};$$

$$x = \textcircled{\text{⊕}} \rightarrow \textcircled{\text{Pfvb}}_{\text{IvIv}}; x = \textcircled{\text{*}} \rightarrow \textcircled{\text{Ifvb}}_{\text{IvIv}}; x = \textcircled{\text{⊃}} \rightarrow \textcircled{\text{Pfvb}}_{\text{PvPv}};$$

$$\text{variable}(x) \rightarrow \text{type}(x); \text{atom}(x) \rightarrow \emptyset;$$

$$a(x) = \textcircled{\text{qu}} \rightarrow \textcircled{\text{Iv}};$$

$$a(x) = \textcircled{\text{pcond}} \vee a(x) = \textcircled{\text{V}} \vee a(x) = \textcircled{\text{E}} \vee a(x) = \textcircled{\text{E!}} \rightarrow \textcircled{\text{Pv}};$$

$$a(x) = \textcircled{\text{cond}} \vee a(x) = \textcircled{\text{1}} \rightarrow \textcircled{\text{Iv}};$$

$$a(x) = \textcircled{\text{λ}} \rightarrow$$

$$[\text{exprtype}(\text{add}(x)) = \textcircled{\text{Pv}} \rightarrow \textcircled{\text{Pfvb}} * \text{maplistcar}(\text{type}, \text{ad}(x));$$

$$\text{exprtype}(\text{add}(x)) = \textcircled{\text{Iv}} \rightarrow \textcircled{\text{Ifvb}} * \text{maplistcar}(\text{type}, \text{ad}(x));$$

$$\textcircled{\text{⊕}} \rightarrow \emptyset];$$

$$a(x) = \textcircled{\text{label}} \rightarrow \text{type}(\text{ad}(x));$$

$$a(\text{exprtype}(a(x))) = \textcircled{\text{Pfvb}} \rightarrow \textcircled{\text{Pv}};$$

$$a(\text{exprtype}(a(x))) = \textcircled{\text{Ifvb}} \rightarrow \textcircled{\text{Iv}};$$

$$\textcircled{\text{⊕}} \rightarrow \emptyset]$$

$$\text{args}(x) ::= d(\text{exprtype}(x))$$

$$\text{newvarex}(x,y) ::= \text{newvart}(\text{exprtype}(x),y)$$

$$\text{mol}(x) ::= \text{atom}(x) \vee a(x) = \textcircled{\text{qu}} \vee \text{variable}(x)$$

$$x \triangleleft y ::= \text{contains}(\text{mol}, x,y)$$

$$x \triangleleft y ::= x \triangleleft y \wedge x \neq y$$

$$x \triangleleft y ::= \text{contains}((\lambda(y) (\text{mol}(y) \vee (\text{listbinder}(a(y)) \wedge x \in \text{ad}(y)))$$

$$\vee (a(y) = \textcircled{\text{label}} \wedge x = \text{ad}(y))))), x,y)$$

$$x \triangleleft y ::= x \triangleleft y \wedge x \neq y$$

$$x \triangleleft^{\text{t}} y ::= [\sim x \triangleleft y \rightarrow \textcircled{\text{⊕}} ; a(x) = \textcircled{\text{label}} \rightarrow \textcircled{\text{⊕}} ; \textcircled{\text{⊕}} \rightarrow x \triangleleft^{\text{t}} a(y) \vee x \triangleleft^{\text{t}} d(y)]$$

$$S(x,y,z) =: Subst(mol,x,y,z)$$

$$Sf(x,y,z) =: Subst((\lambda(u) (mol(u) \vee (listbinder(a(u)) \vee x \in ad(u)) \vee (a(u) = \text{label} \wedge x = ad(u))))), x,y,z)$$

$$Snf(x,y,z) =: Subst((\lambda(u) (mol(u) \vee a(u) = \lambda \vee a(u) = \text{label}))), x,y,z)$$

$$Ssl(x,y,z) =: [atom(x) \rightarrow z ; \oplus \rightarrow S(newvarex(a(x), (x*(y*z))), a(y), Ssl(d(x), d(y), S(a(x), newvarex(a(x), (x*(y*z))), z)))]$$

$$Ssfl(x,y,z) =: [atom(x) \rightarrow z ; \opl� \rightarrow Sf(newvarex(a(x), (x*(y*z))), a(y), Ssfl(d(x), d(y), Sf(a(x), newvarex(a(x), (x*(y*z))), z)))]$$

$$freecheck(x,y,z) =:$$

$$[mol(z) \vee \sim x \triangleleft z \rightarrow \oplus ;$$

$$listbinder(a(z)) \rightarrow andlista((\lambda(u) (\sim u \triangleleft y)), ad(z)) \wedge freecheck(x,y,add(z)) ;$$

$$a(z) = \text{label} \rightarrow \sim ad(z) \triangleleft y \wedge freecheck(x,y, add(z)) ;$$

$$\oplus \rightarrow freecheck(x,y, a(z)) \wedge freecheck(x,y,d(z))]$$

$$formsimp(P_{IV},x) =: [atom(x) \rightarrow Pv(x) \vee x = \oplus \vee x = \opl� ;$$

$$a(x) = \text{pcond} \rightarrow a(ast(x)) = \oplus \wedge andlista((\lambda(y) (formsimp(P_{IV}, a(y)) \wedge formsimp(P_{IV}, ad(y)) \wedge dd(y) = \emptyset)), d(x)) ;$$

$$a(x) = \forall \wedge a(x) = \mathbb{E} \wedge a(x) = \mathbb{E}! \rightarrow (a(x) = \mathbb{E}! \supset dad(x) = \emptyset)$$

$$\wedge andlista(Iv, ad(x)) \wedge formsimp(P_{IV}, add(x)) \wedge ddd(x) = \emptyset ;$$

$$\oplus \rightarrow (Pfatom(a(x)) \vee Pfv(a(x))) \wedge andlistlista((\lambda(u,v) u = exprtype(v)),$$

$$args(a(x), d(x)) \wedge andlista(P_{IV}, d(x))]$$

$$\begin{aligned}
 \text{termsimp}(P_{IV}, x) &=: [\text{atom}(x) \rightarrow Iv(x); a(x) = \text{qu} \wedge dd(x) = \emptyset \rightarrow \text{tr} ; \\
 a(x) &= \text{cond} \rightarrow a(\text{ast}(x)) = \text{tr} \wedge \text{andlista}((\lambda(y) (\text{formsimp}(P_{IV}, a(y)) \\
 &\quad \wedge \text{termsimp}(P_{IV}, ad(y)) \wedge dd(y) = \emptyset)), d(x)) ; \\
 a(x) &= \text{I} \rightarrow Iv(aad(x)) \wedge \text{formsimp}(P_{IV}, add(x)) \wedge dad(x) = \emptyset \wedge ddd(x) = \emptyset ; \\
 \text{tr} &\rightarrow (\text{Ifatom}(a(x)) \vee \text{Ifv}(a(x))) \wedge \text{andlistlista}((\lambda(u,v) u = \text{exprtype}(v)), \\
 &\quad \text{args}(a(x)), d(x)) \wedge \text{andlista}(P_{IV}, d(x))]
 \end{aligned}$$

$$\begin{aligned}
 \text{Simplexpr}(x) &=: \text{Pfatom}(x) \vee \text{Pfv}(x) \vee \text{Ifatom}(x) \vee \text{Ifv}(x) \\
 &\quad \vee \text{formsimp}(\text{Simplexpr}, x) \vee \text{termsimp}(\text{Simplexpr}, x)
 \end{aligned}$$

$$\text{Simpleformula}(x) =: \text{formsimp}(\text{Simplexpr}, x)$$

$$\text{Simpleterm}(x) =: \text{termsimp}(\text{Simplexpr}, x)$$

$$\text{formtofunctiontp}(P, x) =:$$

$$\begin{aligned}
 &\text{Pfatom}(x) \vee \text{Pfv}(x) \vee \text{Ifatom}(x) \vee \text{Ifv}(x) \vee \\
 &(\text{a}(x) = \text{I} \wedge \text{varlist}(ad(x)) \wedge P(add(x)) \\
 &\quad \wedge (\text{exprtype}(add(x)) = \text{Pv} \vee \text{exprtype}(add(x)) = \text{Iv}) \wedge ddd(x) = \emptyset) \vee \\
 &(\text{a}(x) = \text{label} \wedge (\text{Pfv}(ad(x)) \vee \text{Ifv}(ad(x))) \wedge \text{type}(ad(x)) = \text{exprtype}(add(x)) \\
 &\quad \wedge \text{aadd}(x) = \text{I} \wedge \text{varlist}(adadd(x)) \wedge P(addadd(x)) \\
 &\quad \wedge (\text{exprtype}(addadd(x)) = \text{Pv} \vee \text{exprtype}(addadd(x)) = \text{Iv}) \\
 &\quad \wedge ad(x) \trianglelefteq \text{addadd}(x) \wedge \sim ad(x) \trianglelefteq \text{Snf}(ad(x), \emptyset, \text{addadd}(x)) \\
 &\quad \wedge ddd(x) = \emptyset \wedge dddadd(x) = \emptyset)
 \end{aligned}$$

expression(x) =:

formtofunctiontp(*expression*, x) \wedge

[*atom*(x) \rightarrow Pv(x) \vee x = $\textcircled{\oplus}$ \vee x = $\textcircled{\ominus}$ \vee Iv(x)] ;

a(x) = $\textcircled{\text{qu}}$ \wedge dd(x) = $\emptyset \rightarrow \textcircled{\oplus}$;

a(x) = $\textcircled{\text{pcond}}$ \rightarrow a(*ast*(x)) = $\textcircled{\oplus} \rightarrow$ *andlista*((λ (y) (*expression*(a(y))
 \wedge *expression*(ad(y)) \wedge dd(y) = \emptyset

\wedge *exprtype*(a(y)) = $\textcircled{\text{Pv}}$ \wedge *exprtype*(ad(y)) = $\textcircled{\text{Pv}}$)),d(x)) ;

a(x) = $\textcircled{\text{cond}}$ \rightarrow a(*ast*(x)) = $\textcircled{\oplus} \wedge$ *andlista*((λ (y) (*expression*(a(y))

\wedge *expression*(ad(y)) \wedge dd(y) = $\emptyset \wedge$ *exprtype*(a(y)) = $\textcircled{\text{Pv}}$

\wedge *exprtype*(ad(y)) = $\textcircled{\text{Iv}}$)),d(x)) ;

a(x) = $\textcircled{\text{V}}$ \vee a(x) = $\textcircled{\text{E}}$ \vee a(x) = $\textcircled{\text{E!}}$ \rightarrow *andlista*(Iv,ad(x)) \wedge *expression*(add(x))

\wedge (a(x) = $\textcircled{\text{E!}}$ \supset dad(x) = \emptyset) \wedge ddd(x) = $\emptyset \wedge$ *exprtype*(add(x)) = $\textcircled{\text{Pv}}$;

a(x) = $\textcircled{\text{I}}$ \rightarrow Iv(add(x)) \wedge *expression*(add(x)) \wedge dad(x) = \emptyset

\wedge ddd(x) = $\emptyset \wedge$ *exprtype*(add(x)) = $\textcircled{\text{Pv}}$; $\textcircled{\oplus} \rightarrow$

formtofunctiontp(*expression*, a(x)) \wedge *andlistlista*((λ (u,v) u = *exprtype*(v)),

args(a(x),d(x)) \wedge *andlista*(*expression*,d(x))]

4.4 Table 4. Axioms

Predicate calculus axioms

1. $p \supset (q \supset p)$
2. $(s \supset (p \supset q)) \supset ((s \supset p) \supset (s \supset q))$
3. $((p \supset \text{⊥}) \supset \text{⊥}) \supset p$
4. $\forall (x) (p \supset P(x)) \supset (p \supset \forall (x) P(x))$
5. $\forall (x) P(x) \supset P(x)$

6. reflexivity of =

$$x = x$$

7. replaceability of = and ≡

a. $x = y \supset (P(x) \supset P(y))$

b. $(p \equiv q) \supset (P(p) \supset P(q))$

8. Peano axiom 3

$$\sim atom(x*y)$$

9. Peano axiom 4

$$x*u = y*v \supset (x = y \wedge u = v)$$

10. Peano axiom 5, induction

$$(\forall(x) (atom(x) \supset P(x)) \wedge \forall(x,y) ((P(x) \wedge P(y)) \supset P(x*y))) \supset \forall(x) P(x)$$

11. ⊕ definition

$$\oplus \equiv (\oplus \supset \oplus)$$

12. ⊖ definition

$$\exists(x) P(x) \equiv \sim \forall(x) \sim P(x)$$

13. ∃! definition

$$\exists!(x) P(x) \equiv (\exists(x) P(x) \wedge \forall(x,y) ((P(x) \wedge P(y)) \supset x = y))$$

14. ι definition

$$\exists!(x) P(x) \supset P(\iota(x) P(x))$$

15. Disjointness of atom classes

a. $(Iv(x) \vee Pfvb(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Pv(x)$

b. $(Pv(x) \vee Pfvb(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Iv(x)$

c. $(Pv(x) \vee Iv(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Pfvb(x)$

d. $(Pv(x) \vee Iv(x) \vee Pfvb(x) \vee nonvaratom(x)) \supset \sim Ifvb(x)$

16. New variables are variables

- a. $Pv(newpv(x))$
- b. $Iv(newiv(x))$
- c. $Pfvb(newpfvb(x))$
- d. $Ifvb(newifvb(x))$

17. New variables are new

- a. $\sim newpv(x) \triangleleft x$
- b. $\sim newiv(x) \triangleleft x$
- c. $\sim newpfvb(x) \triangleleft x$
- d. $\sim newifvb(x) \triangleleft x$

18. Generatability of certain functions. (See Table 6 for definition of *Pfstep*.)

$$\emptyset \supset Pfstep(x)$$

4.5 Table 5. Rules of Inference

Rules of inference: (T is defined in Table 7)

1. Modus Ponens

$$(T(x \supset y) \wedge T(x)) \supset T(y)$$

2. Generalization

$$(T(y) \wedge \text{Iv}(x) \wedge \sim x \not\Leftarrow y) \supset T(\forall(x) y)$$

3. Change of bound variable

$$(T(y) \wedge \text{variable}(x) \wedge \text{type}(x) = \text{type}(z) \wedge \sim x \Leftarrow u \wedge \sim z \Leftarrow u \\ \wedge S(u, S(x, z, u), y) = S(u, S(x, z, u), v)) \supset T(v)$$

4. Substitution of simple expression for variable

$$(T(y) \wedge \text{variable}(x) \wedge \text{Simplexpr}(z) \wedge \text{type}(x) = \text{exprtype}(z) \\ \wedge \text{freecheck}(x, z, y)) \supset T(Sf(x, z, y))$$

5. Substitution of function expression for function variable

$$(T(y) \wedge T(u) \wedge z \Leftarrow u \wedge (Pfv(x) \vee Ifv(x)) \wedge \text{type}(x) = \text{exprtype}(z) \\ \wedge \text{freecheck}(x, z, y)) \supset T(Sf(x, z, y))$$

6. Application of a λ expression to arguments (where the λ expression is not inside another function expression).

$$(T(u) \wedge \text{expression}(v) \wedge aa(y) = (\lambda \\ \wedge \text{andlistlista}((\lambda(x, z) \text{freecheck}(x, z, \text{adda}(y))), \text{ada}(y), d(y)) \\ \wedge \text{Snf}(y, \text{Ssfl}(\text{ada}(y), d(y), \text{adda}(y)), u) = \text{Snf}(y, \text{Ssfl}(\text{ada}(y), d(y), \text{adda}(y)), v) \\ \wedge (y \Leftarrow u \vee (\text{andlista}((\lambda(z) \sim z \Leftarrow \text{adda}(y)), \text{ada}(y)) \\ \wedge \text{andlistlista}((\lambda(x, z) \text{label} \Leftarrow z \supset x \Leftarrow \text{adda}(y)), \text{ada}(y), d(y)))))) \\ \supset T(v)$$

7. Function recursion

$$\begin{aligned}
 (T(u) \wedge \text{expression}(v) \wedge a(y) = \text{label} \wedge ad(y) \Leftarrow add(y) \\
 \wedge \text{freecheck}(ad(y), y, add(y)) \\
 \wedge \text{Snf}(y, Sf(ad(y), y, add(y)), u) = \text{Snf}(y, Sf(ad(y), y, add(y)), v)) \\
 \supset T(v)
 \end{aligned}$$

8. pcond rule

$$\begin{aligned}
 (\text{Simpleformula}(y) \wedge Pv(w) \\
 \wedge \text{andlista}((\lambda(x) (\text{Simpleformula}(x) \wedge a(x) = \text{pcond} \wedge aad(x) = w)), z) \\
 \wedge u = \text{Ssl}(z, \text{maplistcar}(adad, z), y) \\
 \wedge v = \text{Ssl}(z, \text{maplistcar}((\lambda(x) (\text{pcond} * dd(x))), z), y) \\
 \wedge \sim w \Leftarrow u \wedge \sim w \Leftarrow v) \\
 \supset T(y \Leftarrow ((w \supset u) \wedge ((\sim w) \supset v)))
 \end{aligned}$$

9. cond rule

$$\begin{aligned}
 (\text{Simpleformula}(y) \wedge Pv(w) \\
 \wedge \text{andlista}((\lambda(x) (\text{Simpleterm}(x) \wedge a(x) = \text{cond} \wedge aad(x) = w)), z) \\
 \wedge u = \text{Ssl}(z, \text{maplistcar}(adad, z), y) \\
 \wedge v = \text{Ssl}(z, \text{maplistcar}((\lambda(x) (\text{cond} * dd(x))), z), y) \\
 \wedge \sim w \Leftarrow u \wedge \sim w \Leftarrow v) \\
 \supset T(y \Leftarrow ((w \supset u) \wedge ((\sim w) \supset v)))
 \end{aligned}$$

10. pcond initiation

$$\begin{aligned}
 (\text{Simpleformula}(y) \wedge \text{Simpleformula}(z) \\
 \wedge S(z, (\text{pcond}, (\oplus, z)), u) = S(z, (\text{pcond}, (\oplus, z)), y)) \\
 \supset T(y \Leftarrow u)
 \end{aligned}$$

11. cond initiation

$$\begin{aligned}
 (\text{Simpleformula}(y) \wedge \text{Simpleterm}(z) \\
 \wedge S(z, (\text{cond}, (\oplus, z)), u) = S(z, (\text{cond}, (\oplus, z)), y)) \\
 \supset T(y \Leftarrow u)
 \end{aligned}$$

12. Listbinder notation

$$\begin{aligned}
 & (T(u) \wedge \text{Simpleformula}(x) \wedge (a(x) = \forall \vee a(x) = \exists) \wedge \text{dad}(x) \neq \emptyset \\
 & \wedge \text{Snf}(x, (a(x), \text{aad}(x)), (a(x), \text{dad}(x), \text{add}(x))), u) \\
 & = \text{Snf}(x, (a(x), \text{aad}(x)), (a(x), \text{dad}(x), \text{add}(x))), v) \\
 & \supset T(v)
 \end{aligned}$$

13. Definition of qu

$$T((qu, x) * (qu, y)) = (qu, (x*y))$$

14. Variables are variables

- a. $Pv(x) \supset T(Pv(qu, x))$
- b. $Iv(x) \supset T(Iv(qu, x))$
- c. $Pfvb(x) \supset T(Pfvb(qu, x))$
- d. $Ifvb(x) \supset T(Ifvb(qu, x))$

15. Different atoms are unequal

$$(\text{atom}(x) \wedge \text{atom}(y) \wedge x \neq y) \supset T((qu, x) \neq (qu, y))$$

16. Generation of Predicate function expressions

$$\begin{aligned}
 & (T((z * u) \equiv v) \wedge a(z) = \lambda \wedge z \leq v \wedge \sim z \leq v \\
 & \wedge \text{varlist}(u) \wedge \text{andlista}((\lambda(x) \sim x \leq v), u)) \supset \\
 & T(\text{label}, \text{newvarex}(z, ((z*u) * v)), (\lambda, u) \\
 & S(z, \text{newvarex}(z, ((z*u) * v)), v)) * u)
 \end{aligned}$$

17. Generation of individual function expressions

$$\begin{aligned}
 & (T((z*u) \equiv v) \wedge a(z) = \lambda \wedge z \leq v \wedge \sim z \leq v \\
 & \wedge \text{varlist}(u) \wedge \text{andlista}((\lambda(x) \sim x \leq v), u)) \\
 & \supset T(\text{newiv}((z*u) * v) (\text{newiv}((z*u) * v) \equiv \\
 & (\text{label}, \text{newvarex}(z, ((z*u) * v)), (\lambda, u) \\
 & S(z, \text{newvarex}(z, ((z*u) * v)), v)) * u)
 \end{aligned}$$

18. Dropping one level

$$T(x) \supset T(\underbrace{T((qu, x))}_{\text{down}})$$

19. Raising one level

$$T(\underbrace{T((qu, x))}_{\text{up}}) \supset T(x)$$

4.6 Table 6. Defined Complete Recursive Functions of a Specific Nature

$$Rule_1(x,y,z) =: a(y) = \textcircled{\supset} \wedge z = ad(y) \wedge x = add(y)$$

$$Rule_2(x,y) =: a(x) = \textcircled{\forall} \wedge Iv(aad(x) \wedge dad(x) = \emptyset \wedge y = add(x) \\ \wedge ddd(x) = \emptyset \wedge \sim aad(x) \stackrel{1}{\leq} add(x)$$

$$Rule_3^{\wedge}(v,y,x,z,u) =: variable(x) \wedge type(x) = type(z) \\ \wedge \sim x \leq u \wedge \sim z \leq u \wedge S(u,S(x,z,u),y) = S(u,S(x,z,u),v))$$

$$findx_3(v,y) =: [mol(y) \rightarrow y; a(v) = a(y) \rightarrow findx_3(d(v),d(y)); \\ \textcircled{\oplus} \rightarrow findx_3(a(v),a(y))]$$

$$findz_3(v,y) =: [mol(y) \rightarrow v; \alpha(v) = \alpha(y) \rightarrow findz_3(d(v),d(y)); \\ \textcircled{\oplus} \rightarrow findz_3(a(v),a(y))]$$

$$findu_3(v,y,x) =: [atom(y) \rightarrow \emptyset ; listbinder(a(y)) \wedge ad(y) \neq ad(v) \rightarrow \\ [x \in ad(y) \rightarrow y; \textcircled{\oplus} \rightarrow v]; \\ a(y) - \textcircled{\text{label}} \wedge ad(y) \neq ad(v) \rightarrow [x = ad(y) \rightarrow y; \textcircled{\oplus} \rightarrow v]; \\ findu_3(a(v), a(y), x) \neq \emptyset \rightarrow findu_3(a(v),a(y),x) ; \\ \textcircled{\oplus} \rightarrow findu_3(d(v),d(y),x)]$$

$$Rule_3(v,y) = Rule_3^{\wedge}(v,y, findx_3(v,y), findz_3(v,y), findu_3(v,y, findx_3(v,y)))$$

$$Rule_4^{\wedge}(u,y,x,z) =: variable(x) \wedge simpleexpr(z) \wedge type(x) = exprtype(z) \\ \wedge freecheck(x,z,y) \wedge u = Sf(x,z,y)$$

$$findx_4(u,y) =: findx_3(u,y)$$

$$findz_4(u,y) =: findz_3(u,y)$$

$$Rule_4(u,y) =: Rule_4^{\wedge}(u,y, findx_4(u,y), findz_4(u,y))$$

$$Rule_5^{\wedge}(v,y,u,x,z) =: z \leq u \wedge (Pfv(x) \vee Ifv(x)) \wedge type(x) = exprtype(z) \\ \wedge freecheck(x,z,y) \wedge v = Sf(x,z,y)$$

$$findx_5(v,y) =: findx_3(v,y)$$

$$findz_5(v,y) =: findz_3(v,y)$$

$$Rule_5(v,y,u) =: Rule'_5(v,y,u, findx_5(v,y), findz_5(v,y))$$

$$Rule'_6(v,u,y) =: expression(v) \wedge aa(y) = (\lambda) \wedge$$

$$andlistlista((\lambda(x,z) freecheck(x,z,adda(y))), ada(y), d(y)) \wedge$$

$$Snf(y, Ssfl(ada(y),d(y), adda(y)),u) = Snf(y, Ssfl(ada(y), d(y), adda(y)),v) \wedge$$

$$(y \leq u \vee (andlista((\lambda(z) \sim z \stackrel{!}{\leq} adda(y)), ada(y)))$$

$$\wedge andlistlista((\lambda(x,z) \text{label} \leq z \supset x \leq adda(y)), ada(y), d(y)))$$

$$findy_6(v,u) =: [(atom(u) \wedge atom(v)) \rightarrow \emptyset;$$

$$aa(v) = (\lambda) \wedge u = Ssfl(ada(v), d(v), adda(v)) \rightarrow v;$$

$$aa(u) = (\lambda) \wedge v = Ssfl(ada(u), d(u), adda(u)) \rightarrow u ;$$

$$a(v) \neq a(u) \rightarrow findy_6(a(v), a(u)) ;$$

$$\oplus \rightarrow findy_6(d(v), d(u))]$$

$$Rule_6(v,u) =: Rule'_6(v,u, findy_6(v,u))$$

$$Rule'_7(v,u,y) =: expression(v) \wedge a(y) = \text{label} \wedge ad(y) \leq add(y) \wedge$$

$$freecheck(ad(y), y, add(y))$$

$$\wedge Snf(y, Sf(ad(y), y, add(y)),u) = Snf(y, Sf(ad(y), y, add(y)),v)$$

$$findy_7(v,u) =: [(atom(u) \wedge atom(v)) \rightarrow \emptyset ;$$

$$a(v) = \text{label} \wedge v \neq u \rightarrow v;$$

$$a(u) = \text{label} \wedge u \neq v \rightarrow u;$$

$$a(v) \neq a(u) \rightarrow findy_7(a(v), a(u));$$

$$\oplus \rightarrow findy_7(d(v), d(u))]$$

$$Rule_7(v,u) =: Rule'_7(v,u, findy_7(v,u))$$

$$\begin{aligned}
 Rule_8(x, z, y, w, u, v) &= Simpleformula(y) \wedge Pv(w) \\
 &\wedge andlista((\lambda(x) (Simpleformula(x) \wedge a(x) = \underbrace{pcond}_{\text{pcond}} \wedge aad(x) = w)), z) \\
 &\wedge u = Ssl(z, maplistcar(adad, z), y) \\
 &\wedge v = Ssl(z, maplistcar((\lambda(x) (\underbrace{pcond}_{\text{pcond}} * dd(x))), z), y) \\
 &\wedge \sim w \leq u \wedge \sim w \leq v \wedge x = \underbrace{(\lambda(y) (\underbrace{((\lambda(w) (\underbrace{(\lambda(u) (\underbrace{(\lambda(\sim w) (\lambda(v))})))}_{\text{pcond}}))}_{\text{pcond}}))}_{\text{pcond}}))}_{\text{pcond}}
 \end{aligned}$$

$$findz_8(y, u) = [atom(y) \vee u = y \rightarrow \emptyset;$$

$$a(y) = \underbrace{pcond}_{\text{pcond}} \wedge adad(y) = u \rightarrow (\lambda(y) ());$$

$$\oplus \rightarrow reverseconc(findz_8(a(y), a(u)), findz_8(d(y), d(u)))]$$

$$Rule'_8(x, y, w, u, v) = Rule_8(x, findz_8(y, u), y, w, u, v)$$

$$Rule_8(x) = Rule''_8(x, ad(x), adadadd(x), addadadd(x), addaddadd(x))$$

$$\begin{aligned}
 Rule_9(x, z, y, w, u, v) &= Simpleformula(y) \wedge Pv(w) \\
 &\wedge andlista((\lambda(x) (Simpleterm(x) \wedge a(x) = \underbrace{cond}_{\text{cond}} \wedge aad(x) = w)), z) \\
 &\wedge u = Ssl(z, maplistcar(adad, z), y) \\
 &\wedge v = Ssl(z, maplistcar((\lambda(x) (\underbrace{cond}_{\text{cond}} * dd(x))), z), y) \\
 &\wedge \sim w \leq u \wedge \sim w \leq v \wedge x = \underbrace{(\lambda(y) (\underbrace{((\lambda(w) (\underbrace{(\lambda(u) (\underbrace{(\lambda(\sim w) (\lambda(v))})))}_{\text{cond}}))}_{\text{cond}}))}_{\text{cond}}))}_{\text{cond}}
 \end{aligned}$$

$$findz_9(y, u) = [atom(y) \vee u = y \rightarrow \emptyset;$$

$$a(y) = \underbrace{cond}_{\text{cond}} \wedge adad(y) = u \rightarrow (\lambda(y) ());$$

$$\oplus \rightarrow reverseconc(findz_9(a(y), a(u)), findz_9(d(y), d(u)))]$$

$$Rule'_9(x, y, w, u, v) = Rule_9(x, findz_9(y, u), y, w, u, v)$$

$$Rule_9(x) = Rule''_9(x, ad(x), adadadd(x), addadadd(x), addaddadd(x))$$

$$\begin{aligned}
 Rule_{10}(x, z, y, u) &= Simpleformula(y) \wedge Simpleformula(z) \\
 &\wedge S(z, (\underbrace{(pcond, (\oplus, z))}_{\text{pcond, (\oplus, z)}}), u) = S(z, (\underbrace{(pcond, (\oplus, z))}_{\text{pcond, (\oplus, z)}}), y) \wedge \\
 &x = (y \underbrace{\equiv}_{\text{equiv}} u)
 \end{aligned}$$

$$findz_{10}(y,u) =: [atom(y) \vee u = y \rightarrow \emptyset;$$

$$y = (\text{pcond}, (\oplus, u)) \rightarrow u;$$

$$u = (\text{pcond}, (\oplus, y)) \rightarrow y;$$

$$a(y) = a(u) \rightarrow findz_{10}(d(y), d(u)) ;$$

$$\oplus \rightarrow findz_{10}(a(y), a(u))]$$

$$Rule''_{10}(x,y,u) = Rule'_{10}(x,findz_{10},y,u)$$

$$Rule_{10}(x) = Rule''_{10}(x,ad(x),add(x))$$

$$Rule'_{11}(x,z,y,u) =: Simpleformula(y) \wedge Simpleterm(z)$$

$$\wedge S(z, (\text{cond}, (\oplus, z)), u) = S(z, (\text{cond}, (\oplus, z)), y)$$

$$\wedge x = (y \equiv u)$$

$$findz_{11}(y,u) =: [atom(y) \vee u = y \rightarrow \emptyset;$$

$$y = (\text{cond}, (\oplus, u)) \rightarrow u;$$

$$u = (\text{cond}, (\oplus, y)) \rightarrow y;$$

$$a(y) = a(u) \rightarrow findz_{11}(d(y), d(w));$$

$$\oplus \rightarrow findz_{11}(a(y), a(u))]$$

$$Rule''_{11}(x,y,u) =: Rule'_{11}(x,findz_{11}(y,u),y,u)$$

$$Rule_{11}(x) =: Rule''_{11}(x,ad(x),add(x))$$

$$Rule'_{12}(v,u,x) =: Simpleformula(x) \wedge (a(x) = \forall \vee a(x) = \exists) \wedge dad(x) \neq \emptyset$$

$$\wedge Snf(x, (a(x), (aad(x)), (a(x), dad(x), add(x))), u)$$

$$= Snf(x, (a(x), (aad(x))), (a(x), dad(x), add(x)), v)$$

$$findx_{12}(v,u) =: [atom(u) \vee u = v \rightarrow \emptyset ;$$

$$a(u) = \forall \vee a(u) = \exists \rightarrow [ad(u) = (aad(v)) \rightarrow v;$$

$$ad(v) = (aad(u)) \rightarrow u;$$

$$\oplus \rightarrow findx_{12}(add(v), add(u))]$$

$$a(v) = a(u) \rightarrow findx_{12}(d(v), d(u));$$

$$\oplus \rightarrow findx_{12}(a(v), a(u))]$$

$$Rule_{12}(v,u) =: Rule'_{12}(v,u, findx_{12}(v,u))$$

Rule₁₃(v) =:

$$v = (((qu, adadad(v)) * (qu, adaddad(v))) = (qu, (adadad(v) * adaddad(v))))$$

Rule₁₄(u) =: (u = Pv((qu, adad(u))) ^ Pv(adad(u)))

$$\vee (u = Iv((qu, adad(u))) \wedge Iv(adad(u)))$$

$$\vee (u = Pfvb((qu, adad(u))) \wedge Pfvb(adad(u)))$$

$$\vee (u = Ifvb((qu, adad(u))) \wedge Ifvb(adad(u)))$$

Rule₁₅(u) =: u = ((qu, adad(u)) ≠ (qu, adadd(u)))

$$\wedge atom(adad(u)) \wedge atom(adadd(u)) \wedge adad(u) \neq adadd(u)$$

Rule₁₆[~](w,y,u,v,z) =: y = ((z*u) ≡ v) ^

$$\alpha(x) = (\lambda) \wedge z \leq v \wedge \sim z \leq v \wedge$$

$$varlist(u) \wedge andlista((\lambda(x) \sim x \leq v), u) \wedge$$

$$w = (\exists \supset) ((label, newvarex(z, ((z*u)*v)), (\lambda, u) S(z, newvarex(z, ((z*u)*v)), v)) *u)$$

Rule₁₆(w,y) = Rule₁₆[~](w,y,dad(y),add(y),aad(y))

Rule₁₇[~](w,y,u,v,z) =: y = ((z*u) ≡ v) ^

$$\alpha(z) = (\lambda) \wedge z \leq v \wedge \sim z \leq v \wedge$$

$$varlist(u) \wedge andlista((\lambda(x) \sim x \leq v), u) \wedge$$

$$w = (\exists) (newiv((z*u)*v)) (newiv((z*u)*v) =$$

$$((label, newvarex(z, ((z*u)*v)), (\lambda, u) S(z, newvarex(z, ((z*u)*v)), v)) *u)$$

$$S(z, newvarex(z, ((z*u)*v)), v)) *u)$$

Rule₁₇(w,y) =: Rule₁₇[~](w,y, dad(y), add(y), aad(y))

$Pfstep(x) =: \sim atom(x) \wedge$

$(orlistcar((\lambda(y) orlistcar((\lambda(z) Rule_1(a(x), z, y)), d(x))), d(x))$
 $\vee orlistcar((\lambda(y) Rule_2(a(x), y)), d(x))$
 $\vee orlistcar((\lambda(y) Rule_3(a(x), y)), d(x))$
 $\vee orlistcar((\lambda(y) Rule_4(a(x), y)), d(x))$
 $\vee orlistcar((\lambda(y) orlistcar((\lambda(z) Rule_5(a(x), z, y)), d(x))), d(x))$
 $\vee orlistcar((\lambda(y) Rule_6(a(x), y)), d(x))$
 $\vee orlistcar((\lambda(y) Rule_7(a(x), y)), d(x))$
 $\vee Rule_8(a(x))$
 $\vee Rule_9(a(x))$
 $\vee Rule_{10}(a(x))$
 $\vee Rule_{11}(a(x))$
 $\vee orlistcar((\lambda(y) Rule_{12}(a(x), y)), d(x))$
 $\vee Rule_{13}(a(x))$
 $\vee Rule_{14}(a(x))$
 $\vee Rule_{15}(a(x))$
 $\vee orlistcar((\lambda(y) Rule_{16}(a(x), y)), d(x))$
 $\vee orlistcar((\lambda(y) Rule_{17}(a(x), y)), d(x))$)

$Pcaxiom(x) =:$

$x = p \supset (q \supset p)$
 $x = (s \supset (p \supset q)) \supset ((s \supset p) \supset (s \supset q))$
 $x = ((p \supset \textcircled{p}) \supset \textcircled{p}) \supset p$
 $x = \forall(x) (p \supset P(x)) \supset (p \supset \forall(x) P(x))$
 $x = \forall(x) P(x) \supset P(x)$

Eqaxiom(x) =:

- x = x = x
- ✓ x = x = y \supset (P(x) \supset P(y))
- ✓ x = (p \equiv q) \supset (P(p) \supset P(q))

Peanoaxiom(x) =:

- x = \sim atom(x*y)
- ✓ x = x*u = y*v \supset (x = y \wedge u = v)
- ✓ x = (\forall (x) (atom(x) \supset P(x)) \wedge \forall (x,y) ((P(x) \wedge P(y)) \supset P(x*y))) \supset \forall (x) P(x)

Definitionaxioms(x) =:

- x = $\oplus \equiv (\oplus \supset \oplus)$
- ✓ x = \exists (x) P(x) $\equiv \sim \forall$ (x) \sim P(x)
- ✓ x = $\exists!$ (x) P(x) \equiv (\exists (x) P(x) \wedge \forall (x,y) ((P(x) \wedge P(y)) \supset x = y))
- ✓ x = $\exists!$ (x) P(x) \supset P(ι (x) P(x))

Atomkindaxiom(x) =:

- x = (Iv(x) \vee Pfvb(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Pv(x)
- ✓ x = (Pv(x) \vee Pfvb(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Iv(x)
- ✓ x = (Pv(x) \vee Iv(x) \vee Ifvb(x) \vee nonvaratom(x)) \supset \sim Pfvb(x)
- ✓ x = (Pv(x) \vee Iv(x) \vee Pfvb(x) \vee nonvaratom(x)) \supset \sim Ifvb(x)
- ✓ x = Pv(newpv(x))
- ✓ x = Iv(newiv(x))
- ✓ x = Pfvb(newpfvb(x))
- ✓ x = Ifvb(newifvb(x))
- ✓ x = \sim newpv(x) \Leftarrow x
- ✓ x = \sim newiv(x) \Leftarrow x
- ✓ x = \sim newpfvb(x) \Leftarrow x
- ✓ x = \sim newifvb(x) \Leftarrow x

$$Axiom(x) = Pexiom(x) \vee Eqaxiom(x) \vee Peanoaxiom(x) \vee$$

$$Definitionaxiom(x) \vee Atomkindaxiom(x) \vee$$

$$x = \underbrace{(\emptyset \supset Pfstep(x))}$$

$$Proof(x) =: (Pfstep(x) \vee Axiom(a(x))) \wedge \sim atom(x)$$

$$\wedge (d(x) = \emptyset \vee Proof(d(x)))$$

4.7 Table 7. Definition of T and Immediate Consequences

$$T(x) =: \exists(y) (Proof(y) \wedge a(y) = x)$$

Theorems in the form of rules of inference:

1. $(T(y) \wedge T(z) \wedge Rule_1(x,y,z)) \supset T(x)$
2. $(T(y) \wedge Rule_2(x,y)) \supset T(x)$
3. $(T(y) \wedge Rule_3(x,y)) \supset T(x)$
4. $(T(y) \wedge Rule_4(x,y)) \supset T(x)$
5. $(T(y) \wedge T(z) \wedge Rule_5(x,y,z)) \supset T(x)$
6. $(T(y) \wedge Rule_6(x,y)) \supset T(x)$
7. $(T(y) \wedge Rule_7(x,y)) \supset T(x)$
8. $Rule_8(x) \supset T(x)$
9. $Rule_9(x) \supset T(x)$
10. $Rule_{10}(x) \supset T(x)$
11. $Rule_{11}(x) \supset T(x)$
12. $(T(y) \wedge Rule_{12}(x,y)) \supset T(x)$
13. $Rule_{13}(x) \supset T(x)$
14. $Rule_{14}(x) \supset T(x)$
15. $Rule_{15}(x) \supset T(x)$
16. $(T(y) \wedge Rule_{16}(x,y)) \supset T(x)$
17. $(T(y) \wedge Rule_{17}(x,y)) \supset T(y)$

4.8 Table 8. Non-Recursive Definitions Especially Useful for Meta-Theorems. Some Immediate Consequences.

$$Pfe(x) =: a(exprtype(x)) = \textcircled{Pfvb} \wedge \mathfrak{H}(y) (T(y) \wedge x \triangleleft y)$$

$$Ife(x) =: a(exprtype(x)) = \textcircled{Ifvb} \wedge \mathfrak{H}(y) (T(y) \wedge x \triangleleft y)$$

$$function(x) =: Pfe(x) \vee Ife(x)$$

$$wfexpression(x) =:$$

$$function(x) \vee [atom(x) \rightarrow Pv(x) \vee x = \textcircled{\oplus} \vee x = \textcircled{\ominus} \vee Iv(x)] ;$$

$$a(x) = \textcircled{qu} \wedge dd(x) = \emptyset \rightarrow \textcircled{\oplus} ;$$

$$a(x) = \textcircled{pcond} \rightarrow a(ast(x)) = \textcircled{\oplus} \wedge andlista((\lambda(y) (wfexpression(a(y)) \wedge wfexpression(ad(y)) \wedge dd(y) = \emptyset \wedge exprtype(a(y)) = \textcircled{Pv} \wedge exprtype(ad(y)) = \textcircled{Pv})), d(x)) ;$$

$$a(x) = \textcircled{cond} \rightarrow a(ast(x)) = \textcircled{\oplus} \wedge andlista((\lambda(y) (wfexpression(a(y)) \wedge wfexpression(ad(y)) \wedge dd(y) = \emptyset \wedge exprtype(a(y)) = \textcircled{Pv} \wedge exprtype(ad(y)) = \textcircled{Iv})), d(x)) ;$$

$$a(x) = \textcircled{\forall} \vee a(x) = \textcircled{\mathfrak{H}} \vee a(x) = \textcircled{\mathfrak{H}!} \rightarrow andlista(Iv, ad(x)) \wedge wfexpression(add(x)) \wedge (a(x) = \textcircled{\mathfrak{H}!} \supset dad(x) = \emptyset) \wedge ddd(x) = \emptyset \wedge exprtype(add(x)) = \textcircled{Pv} ;$$

$$a(x) = \textcircled{\exists} \rightarrow Iv(aad(x)) \wedge wfexpression(add(x)) \wedge dad(x) = \emptyset \wedge ddd(x) = \emptyset \wedge exprtype(add(x)) = \textcircled{Pv} ;$$

$$\textcircled{\oplus} \rightarrow function(a(x)) \wedge andlistlista((\lambda(u,v) u = exprtype(v)), args(a(x)), d(x)) \wedge andlista(wfexpression, d(x))]$$

$$F(x) = wfexpression(x) \wedge exprtype(x) = \textcircled{Pv}$$

$$Tm(x) = wfexpression(x) \wedge exprtype(x) = \textcircled{Iv}$$

$$form(x) =: F(x) \vee Tm(x)$$

$$wffragment(x) =: \mathfrak{H}(y) (wfexpression(y) \wedge x \triangleleft y)$$

Theorems:

$$wfexpression(x) \equiv (form(x) \vee function(x))$$

$$F(x) \equiv [atom(x) \rightarrow Pv(x) \vee x = \textcircled{\text{P}} \vee x = \textcircled{\text{Q}} ;$$

$$a(x) = \textcircled{\text{pcond}} \rightarrow a(ast(x)) = \textcircled{\text{P}} \wedge andlista((\lambda(y) (F(a(y)) \wedge F(ad(y)) \wedge dd(y) = \emptyset)), d(x)) ;$$

$$a(x) = \textcircled{\text{V}} \vee a(x) = \textcircled{\text{H}} \vee a(x) = \textcircled{\text{H}!} \rightarrow andlista(Iv, ad(x)) \wedge F(add(x)) \wedge (a(x) = \textcircled{\text{H}!} \supset dad(x) = \emptyset) \wedge ddd(x) = \emptyset ;$$

$$\textcircled{\text{P}} \rightarrow Pfe(a(x)) \wedge andlistlista((\lambda(u,v) u = exprtype(v)), args(a(x)), d(x)) \wedge andlista(wfexpression, d(x))]$$

$$Tm(x) \equiv [atom(x) \rightarrow Iv(x) ; a(x) = \textcircled{\text{qu}} \wedge dd(x) = \emptyset \rightarrow \textcircled{\text{P}} ;$$

$$a(x) = \textcircled{\text{cond}} \rightarrow a(ast(x)) = \textcircled{\text{P}} \wedge andlista((\lambda(y) (F(a(y)) \wedge Tm(ad(y)) \wedge dd(y) = \emptyset)), d(x)) ;$$

$$a(x) = \textcircled{\text{I}} \rightarrow Iv(aad(x)) \wedge F(add(x)) \wedge dad(x) = \emptyset \wedge ddd(x) = \emptyset ;$$

$$\textcircled{\text{P}} \rightarrow Ife(a(x)) \wedge andlistlista((\lambda(u,v) u = exprtype(v)), args(a(x)), d(x)) \wedge andlista(wfexpression, d(x))]$$

$$Pfe(x) \equiv (Pfatom(x) \vee Pfv(x) \vee$$

$$(a(x) = \textcircled{\lambda} \wedge varlist(ad(x)) \wedge F(add(x)) \wedge ddd(x) = \emptyset) \vee$$

$$(a(x) = \textcircled{\text{label}} \wedge Pfv(ad(x)) \wedge Pfe(add(x)) \wedge ad(x) \triangleleft add(x) \wedge$$

$$\sim ad(x) \triangleleft Snf(ad(x), \emptyset, add(x)) \wedge ddd(x) = \emptyset \wedge$$

$$type(ad(x)) = exprtype(add(x)) \wedge \textcircled{\text{H}}(y) (T(y) \wedge x \triangleleft y)))$$

$$Ife(x) \equiv (Ifatom(x) \vee Ifv(x) \vee$$

$$(a(x) = \textcircled{\lambda} \wedge varlist(ad(x)) \wedge Tm(add(x)) \wedge ddd(x) = \emptyset) \vee$$

$$(a(x) = \textcircled{\text{label}} \wedge Ifv(ad(x)) \wedge Ife(add(x)) \wedge ad(x) \triangleleft add(x) \wedge$$

$$\sim ad(x) \triangleleft Snf(ad(x), \emptyset, add(x)) \wedge ddd(x) = \emptyset \wedge$$

$$type(ad(x)) = exprtype(add(x)) \wedge \textcircled{\text{H}}(y) (T(y) \wedge x \triangleleft y)))$$

$$(T(y) \wedge \text{variable}(x) \wedge \text{wfexpression}(z) \wedge \text{type}(x) = \text{exprtype}(z) \wedge \\ \text{freecheck}(x,y,z)) \supset T(Sf(x,z,y))$$

The above theorem, or meta-theorem, summarizes Rules 4 and 5. This could be used as a rule, but unlike our original 19 rules, this rule contains function expressions which are not complete recursing. For that reason this rule is not equivalent to Rules 4 and 5 as a rule; it is only equivalent as a meta-theorem.

Let us combine the above theorems on *Pfe* and *Ife* and make some minor changes to obtain:

$$\begin{aligned} \text{function}(x) &\equiv \text{Pfatom}(x) \vee \text{Pfv}(x) \vee \text{Ifatom}(x) \vee \text{Ifv}(x) \vee \\ (a(x) &= \lambda \wedge \text{varlist}(ad(x)) \wedge \text{form}(add(x)) \wedge \text{ddd}(x) = \emptyset) \vee \\ (a(x) &= \text{label} \wedge (\text{Pfv}(ad(x)) \vee \text{Ifv}(ad(x))) \wedge \text{type}(ad(x)) = \text{exprtype}(add(x)) \wedge \\ aadd(x) &= \lambda \wedge \text{varlist}(adadd(x)) \wedge \text{form}(addadd(x)) \wedge \\ ad(x) &\triangleleft addadd(x) \wedge \sim ad(x) \triangleleft \text{Snf}(ad(x), \emptyset, addadd(x)) \wedge \\ ddd(x) &= \emptyset \wedge \text{dddadd}(x) = \emptyset \quad \wedge \quad \exists(y) (T(y) \wedge x \triangleleft y)) \end{aligned}$$

Note the similarity of this theorem to the definition statement of *formtofunctiontp*. In fact, the similarity is shown by the following theorem:

$$\begin{aligned} \text{function}(x) &\equiv (\text{formtofunctiontp}(\text{wfexpression}, x) \wedge \\ a(x) &= \text{label} \supset \exists(y) (T(y) \wedge x \triangleleft y)) \end{aligned}$$

Bearing this theorem in mind, we see that *expression* and *wfexpression* are almost identical, the only difference being that the $\exists(y)(T(y) \wedge x \triangleleft y)$ condition is added in a couple of places in *wfexpression*. The following theorem indicates this relationship.

$$\begin{aligned} \text{wfexpression}(x) &\equiv (\text{expression}(x) \wedge \forall(z) ((z \triangleleft x \wedge a(z) = \text{label})) \\ &\quad \supset \exists(y) (T(y) \wedge z \triangleleft y)) \end{aligned}$$

Unlike *wfexpression*, *expression* is recursive.

We have the theorem

$$wfexpression(x) \supset expression(x) \quad .$$

We now define:

$$Pfetp(x) := expression(x) \wedge a(exprtype(x)) = \textcircled{Pfvb}$$

$$Ifetp(x) := expression(x) \wedge a(exprtype(x)) = \textcircled{Ifvb}$$

$$functiontp(x) := Pfetp(x) \vee Ifetp(x)$$

$$Ftp(x) := expression(x) \wedge exprtype(x) = \textcircled{Pv}$$

$$Tmtp(x) := expression(x) \wedge exprtype(x) = \textcircled{Iv}$$

$$formtp(x) := Ftp(x) \vee Tmtp(x)$$

And now we have these theorems:

$$functiontp(x) \equiv formtp(x)$$

$$function(x) \equiv (functiontp(x) \wedge \exists(y) (T(y) \wedge x \leq y))$$

$$wfexpression(x) \equiv (expression(x) \wedge$$

$$\forall(z) ((z \leq x \wedge functiontp(z)) \supset \exists(y) (T(y) \wedge z \leq y)))$$

$$wfexpression(x) \equiv (expression(x) \wedge \exists(y) (T(y) \wedge x \leq y))$$

$$form(x) \equiv (formtp(x) \wedge \exists(y) (T(y) \wedge x \leq y))$$

$$F(x) \equiv (Ftp(x) \wedge \exists(y) (T(y) \wedge x \leq y))$$

$$Tm(x) \equiv (Tmtp(x) \wedge \exists(y) (T(y) \wedge x \leq y))$$

$$Pfe(x) \equiv Pfetp(x) \wedge \exists(y) (T(y) \wedge x \leq y)$$

$$Ife(x) \equiv Ifetp(x) \wedge \exists(y) (T(y) \wedge x \leq y)$$

4.9 Table 9. Definitions for Handling Recursive Functions; *apl*

$x \equiv y =: [mol(x) \rightarrow x = y ;$

$listbinder(a(x)) \rightarrow a(x) = a(y) \wedge Ssfl(ad(x), ad(y), add(y)) \equiv add(x) ;$

$a(x) = \underbrace{\text{label}} \rightarrow a(x) = a(y) \wedge Sf(ad(x), ad(y), add(y)) \equiv add(x) ;$

$\textcircled{\text{v}} \rightarrow andlistlista(\equiv, x, y)]$

$replacefx(v, y, u) =: [atom(v) \rightarrow u ;$

$a(v) \Leftarrow y \rightarrow replacefx(d(v), y, Sf(a(v), newvarex(a(v), y*u), u)) ;$

$\textcircled{\text{v}} \rightarrow replacefx(d(v), y, u)]$

$freefix(x, y, z) =:$

$[mol(z) \vee \sim x \Leftarrow z \rightarrow z ;$

$listbinder(a(z)) \rightarrow a(z) *replacefx(ad(z), y, freefix(x, y, d(z))) ;$

$a(z) = \underbrace{\text{label}} \rightarrow a(z) *replacefx(\underbrace{\text{Q}}_{ad(z)} \underbrace{\text{Q}}_{d(z)}, y, freefix(x, y, d(z))) ;$

$\textcircled{\text{v}} \rightarrow freefix(x, y, a(z)) * freefix(x, y, d(z))]$

$freefixlists(x, y, z) =:$

$[atom(x) \rightarrow z ; \textcircled{\text{v}} \rightarrow freefixlists(d(x), y, freefix(a(x), y, z))]$

$Sffix(x, y, z) =: Sf(x, y, freefix(x, y, z))$

$Ssffixl(x, y, z) =: Ssfl(x, y, freefixlists(x, y, z))$

$aploneatom(x) =: [aad(x) \neq qu \rightarrow x ;$

- $a(x) = Pv \rightarrow [Pv(adad(x)) \rightarrow \textcircled{\text{P}} ; \textcircled{\text{P}} \rightarrow \textcircled{\text{P}}] ;$
 - $a(x) = Iv \rightarrow [Iv(adad(x)) \rightarrow \textcircled{\text{I}} ; \textcircled{\text{I}} \rightarrow \textcircled{\text{I}}] ;$
 - $a(x) = Pfvb \rightarrow [Pfvb(adad(x)) \rightarrow \textcircled{\text{P}} ; \textcircled{\text{P}} \rightarrow \textcircled{\text{P}}] ;$
 - $a(x) = Ifvb \rightarrow [Ifvb(adad(x)) \rightarrow \textcircled{\text{I}} ; \textcircled{\text{I}} \rightarrow \textcircled{\text{I}}] ;$
 - $a(x) = newpv \rightarrow (qu, newpv(adad(x))) ;$
 - $a(x) = newiv \rightarrow (qu, newiv(adad(x))) ;$
 - $a(x) = newpfb \rightarrow (qu, newpfb(adad(x))) ;$
 - $a(x) = newifvb \rightarrow (qu, newifvb(adad(x))) ;$
 - $a(x) = a \rightarrow (qu, a(adad(x))) ;$
 - $a(x) = d \rightarrow (qu, d(adad(x))) ;$
- $\textcircled{\text{P}} \rightarrow x]$

$apltwoatoms(x) =:$

- $[a(x) = \textcircled{\supset} \rightarrow [ad(x) = \textcircled{\text{P}} \vee add(x) = \textcircled{\text{P}} \rightarrow \textcircled{\text{P}} ; ad(x) = \textcircled{\text{P}} \rightarrow add(x) ; \textcircled{\text{P}} \rightarrow x] ;$
 - $aad(x) \neq \textcircled{\text{qu}} \vee aadd(x) \neq \textcircled{\text{qu}} \rightarrow x ;$
 - $a(x) = \textcircled{=} \rightarrow [adad(x) = adadd(x) \rightarrow \textcircled{\text{P}} ; \textcircled{\text{P}} \rightarrow \textcircled{\text{P}}] ;$
 - $a(x) = \textcircled{*} \rightarrow ((qu, (adad(x) * adadd(x))) ;$
- $\textcircled{\text{P}} \rightarrow x]$

$$\text{oneapl}(x) =: [\text{mol}(x) \rightarrow x; a(x) = \text{T} \rightarrow \text{T}(\text{oneapl}(ad(x))) \text{J}];$$

$$a(x) = \text{pcond} \vee a(x) = \text{cond} \rightarrow$$

$$[\text{atom}(d(x)) \rightarrow x ;$$

$$aad(x) = \text{E} \rightarrow adad(x) ;$$

$$aad(x) = \text{E} \rightarrow a(x) * dd(x) ;$$

$$\text{E} \rightarrow a(x) * (\text{J} \text{oneapl}(aad(x)) \text{J} adad(x) \text{J} * dd(x))] ;$$

$$a(x) = \text{D} \vee a(x) = \text{E} \vee a(x) = \text{*} \rightarrow$$

$$\text{apltwoatoms} (\text{J} a(x) \text{J} \text{oneapl}(ad(x)) \text{J} \text{oneapl}(add(x)) \text{J}) ;$$

$$a(x) = \text{a} \vee a(x) = \text{d} \vee \text{P} \text{fatom}(a(x)) \vee \text{I} \text{fatom}(a(x)) \rightarrow$$

$$\text{aploneatom} (\text{J} a(x) \text{J} \text{oneapl}(ad(x)) \text{J}) ;$$

$$\text{atom}(a(x)) \rightarrow x ;$$

$$aa(x) = \text{\lambda} \rightarrow \text{Ssffixl}(ada(x), d(x), adda(x)) ;$$

$$aa(x) = \text{label} \rightarrow \text{Sffix}(ada(x), a(x), adda(x)) * d(x) ;$$

$$\text{E} \rightarrow x]$$

$$\text{apl}(x) =: [\text{oneapl}(x) = x \rightarrow x ; \text{E} \rightarrow \text{apl}(\text{oneapl}(x))]$$

4.10 Examples

4.10.1 Example 1. Reflexivity and Transitivity of =

- | | |
|--|------------------------------|
| 1. $x = x$ | Axiom 6 |
| 2. $((\lambda(y)y = x),x)$ | Rule 6 |
| 3. $x = y \supset ((\lambda(y)y = x),x) \supset ((\lambda(y)y = x),y)$ | Axiom 7, Rule 5 |
| 4. $x = y \supset (x = x \supset y = x)$ | Rule 6 |
| 5. $x = y \supset y = x$ | by P.C. |
| 6. $y = z \supset (((\lambda(y)y = x),y) \supset ((\lambda(y)y = x),z))$ | Axiom 7, Rule 5 |
| 7. $y = z \supset (y = x \supset z = x)$ | Rule 6 |
| 8. $z = y \supset (y = x \supset z = x)$ | From Lines 5 & 7,
by P.C. |

In future examples, where I use substitution instances of Lines 5 & 8 followed by P.C. operations, I will merely write the result with the comment "properties of = ."

4.10.2 Example 2. Proof of $\sim atom(x) \equiv x = a(x) * d(x)$
(basic theorem for a and d).

$conseedpred(x) =: \sim atom(x) \supset \exists(y) \exists(z) y*z = x$

- | | |
|---|-----------------|
| 1. $p \supset (\sim p \supset q)$ | By P.C. |
| 2. $f(x) \supset (\sim f(x) \supset \exists(y) \exists(z) y*z = x)$ | By Rule 4 twice |
| 3. $atom(x) \supset (\sim atom(x) \supset \exists(y) \exists(z) y*z = x)$ | By Rule 5 |
| 4. $atom(x) \supset conseedpred(x)$ | By Rule 6 |
| 5. $\forall(x) (atom(x) \supset conseedpred(x))$ | By Rule 2 |

Since $conseedpred$ does not recurse, we were able to generate it without use of Rules 16 or 17. In future examples I will not bother to explicitly generate such functions since it is always trivial via Rule 6.

The axioms and rules used to generate Line 3 are those of the Predicate calculus, followed by Rules 4 and 5. In the future, statements so simply derived will not be explicitly derived but will be merely stated with the comment "by P.C."

6. $u * v = u * v.$ Axiom 6
7. $\exists(y) \exists(z) y * z = u * v$ By P.C.
8. $consedpred(u * v)$ By P.C. & Rule 6
9. $\forall(x,y) (consedpred(x) \wedge consedpred(y)) \supset consedpred(x * y)$ By P.C.
10. $\forall(x)consedpred(x)$ By substitution of *consedpred* for P
in Axiom 10; and P.C. with Lines 5 & 9
11. $\sim atom(u) \supset \exists(x) \exists(z) x * z = u$ By P.C. & Rule 6
12. $\exists(z) x * z = u \supset ((atom(u) \wedge x = u) \vee \exists(z) x * z = u)$ By P.C.
13. $\exists(x) \exists(z) x * z = u \supset \exists(x) ((atom(u) \wedge x = u) \vee \exists(z) x * z = u)$ By P.C.
14. $\sim atom(u) \supset \exists(x) ((atom(u) \wedge x = u) \vee \exists(z) x * z = u)$ From Lines 11 &
13 by P.C.
- $tempone_u(x) = ((atom(u) \wedge x = u) \vee \exists(z) x * z = u)$
15. $\sim atom(u) \supset \exists(x) tempone_u(x)$ Rule 6
16. $(\sim atom(u) \wedge tempone_u(x) \wedge tempone_u(y)) \supset (\exists(z) x * z = u \wedge$
 $\exists(y) y * z = u)$ By P.C. &
Rule 6
17. $(x * z = u \wedge y * w = u) \supset x * z = y * w$ Properties of =
18. $x * z = y * w \supset x = y$ Axiom 9
19. $(\exists(z) x * z = u \wedge \exists(z) y * z = u) \supset x = y$ From Lines 17 & 18,
by P.C.
20. $(\sim atom(u) \wedge tempone_u(x) \wedge tempone_u(y)) \supset x = y$ From Lines 16 & 19
21. $\sim atom(u) \supset \forall(x,y) ((tempone_u(x) \wedge tempone_u(y)) \supset x = y)$ By P.C.

22. $\sim atom(u) \supset \exists!(x) \text{ tempone}_u(x)$ From Axiom 13 and Lines 15 & 21 by P.C.
23. $\sim atom(u) \supset \text{tempone}_u(y) \text{ tempone}_u(y)$ From Axiom 13 and Line 22 by P.C.
24. $\sim atom(u) \supset \text{tempone}_u(a(u))$ Rule 6 twice
25. $\sim atom(u) \supset \exists(z) a(u) * z = u$ Rule 6 & P.C.

We can, from Line 11, similarly derive

26. $\sim atom(u) \supset \exists(z) z * d(u) = u$.
27. $(a(u) * w = u \wedge z * d(u) = u) \supset a(u) = z$ Properties of = and Axiom 9
28. $(a(u) * w = u \wedge z * d(u) = u) \supset a(u) * d(u) = u$ From Line 27 by P.C.
29. $(\exists(z) a(u) * z = u \wedge \exists(z) z * d(u) = u) \supset u = a(u) * d(u)$ From Line 28 by P.C.
30. $\sim atom(u) \supset u = a(u) * d(u)$ From Lines 25, 26, & 29
31. $\sim atom(x) \supset x = a(x) * d(x)$
32. $x = a(x) * d(x) \supset (atom(x) \supset atom(a(x) * d(x)))$ Axiom 7
33. $\sim atom(a(x) * d(x))$ Axiom 8
34. $x = a(x) * d(x) \supset \sim atom(x)$ From Lines 32 & 33 by P.C.
35. $\sim atom(x) \equiv x = a(x) * d(x)$ From Lines 31 & 35

4.10.3 Example 3. Some More Theorems About a and d ;
an Alternative Induction Axiom.

By a technique similar to that of Example 2, we can prove

36. $atom(x) \equiv x = a(x)$ and

37. $atom(x) \equiv x = d(x)$

Corollaries to Line 35 are:

38. $x*y = a(x*y) *d(x*y)$ Substitute $x*y$ for x
in Line 35 & use Axiom 8

39. $x = a(x*y)$ Axiom 9

40. $y = d(x*y)$ Axiom 9

41. $x*y = u \supset (x = a(x*y) \supset x = a(u))$ Axiom 7

42. $u = x*y \supset x = a(u)$ From Lines 39 & 41

43. $u = x*y \supset y = d(u)$ Similarly
all these theorems will be referred to below as
simple properties of a & d .

In view of 42. and 43., we can show the following:

44. $u = x*y \supset (((P(a(u)) \wedge P(d(u))) \supset P(u)) \supset ((P(x) \wedge P(y)) \supset P(x*y)))$

Via Axiom 7

45. $((P(a(u)) \wedge P(d(u))) \supset P(u)) \supset (u = x*y \supset ((P(x) \wedge P(y)) \supset P(x*y)))$

By P.C.

46. $\forall(u) ((P(a(u)) \wedge P(d(u)) \supset P(u)) \supset (u = x*y \supset ((P(x) \wedge P(y)) \supset P(x*y))))$

By P.C.

47. $u = x*y \supset (\forall(u) ((P(a(u)) \wedge P(d(u))) \supset P(u)) \supset ((P(x) \wedge P(y)) \supset P(x*y)))$

By P.C.

48. $\exists(u) u = x*y \supset (\forall(u) ((P(a(u)) \wedge P(d(u))) \supset P(u)) \supset ((P(x) \wedge P(y)) \supset P(x*y)))$

By P.C.

49. $x*y = x*y$ Axiom 6

50. $\exists(u) u = x*y$ From 49 via P.C.

$$51. \quad \forall(u) ((P(a(u)) \wedge P(d(u))) \supset P(u)) \supset ((P(x) \wedge P(y)) \supset P(x*y))$$

From Lines 50 & 48

$$52. \quad \forall(x) ((P(a(x)) \wedge P(d(x))) \supset P(x)) \supset \forall(x,y) ((P(x) \wedge P(y)) \supset P(x*y))$$

from this and Axiom 10, we get an alternative induction axiom

$$53. \quad (\forall(x) (atom(x) \supset P(x)) \wedge \forall(x) ((P(a(x)) \wedge P(d(x))) \supset P(x))) \supset \forall(x)P(x)$$

4.10.4 Example 4. Course of Values Induction (and a Corollary):

Prove ($\forall(x) (\forall(z) (z \triangleleft x \supset P(z)) \supset P(x)) \supset (\forall(x) P(x))$)

Define, for this example

$$B(x) ::= \forall(z) (z \triangleleft x \supset P(z))$$

$$\phi(x) ::= \forall(z) (z \triangleleft x \supset P(z)) \supset P(x)$$

Problem is to prove ($\forall(x) \phi(x) \supset (\forall(x) P(x))$).

(Note: $mol(x) \supset (\phi(x) \equiv P(x))$ holds.)

The proof proceeds as follows:

$$54. \quad z \triangleleft x \supset (z \triangleleft x \vee z = x)$$

$$55. \quad z \triangleleft x \supset (z \triangleleft a(x) \vee z \triangleleft d(x))$$

From defn. of *contains*

$$56. \quad B(x) \supset (z \triangleleft x \supset P(z))$$

$$57. \quad (B(a(x)) \wedge B(d(x))) \supset ((z \triangleleft a(x) \supset P(z)) \wedge (z \triangleleft d(x) \supset P(z)))$$

$$58. \quad (B(a(x)) \wedge B(d(x))) \supset ((z \triangleleft a(x) \vee z \triangleleft d(x)) \supset P(z))$$

$$59. \quad (B(a(x)) \wedge B(d(x))) \supset (z \triangleleft x \supset P(z))$$

From 55 and 58

$$60. \quad (B(a(x)) \wedge B(d(x))) \supset \forall(z) (z \triangleleft x \supset P(z))$$

$$61. \quad (\phi(x) \wedge B(a(x)) \wedge B(d(x))) \supset P(x)$$

$$62. \quad P(x) \supset (z = x \supset P(z))$$

$$63. \quad (B(a(x)) \wedge B(d(x)) \wedge P(x)) \supset ((z \triangleleft x \supset P(z)) \wedge (z = x \supset P(z)))$$

From 59 and 62

64. $((z \triangleleft x \supset P(z)) \wedge (z = x \supset P(z))) \supset (z \triangleleft x \supset P(z))$ From 54
65. $(B(a(x)) \wedge B(d(x)) \wedge P(x)) \supset (z \triangleleft x \supset P(z))$ From 59, 62, & 64
66. $(B(a(x)) \wedge B(d(x)) \wedge P(x)) \supset B(x)$
67. $\Phi(x) \supset ((B(a(x)) \wedge B(d(x))) \supset B(x))$ From 61 and 66
68. $(\forall(x)\Phi(x)) \supset (\forall(x) ((B(a(x)) \wedge B(d(x))) \supset B(x)))$
69. $atom(x) \supset \sim z \triangleleft x$ From definition of *contains*
70. $atom(x) \supset (z \triangleleft x \supset P(z))$
71. $atom(x) \supset (\forall(z) (z \triangleleft x \supset P(z)))$
72. $(\Phi(x) \wedge atom(x)) \supset P(x)$
73. $(atom(x) \wedge P(x)) \supset (z \triangleleft x \supset P(z))$ From 62, 64, and 70
74. $(atom(x) \wedge P(x)) \supset B(x)$
75. $\Phi(x) \supset (atom(x) \supset B(x))$ From 72 and 74
76. $(\forall(x) \Phi(x)) \supset (\forall(x) (atom(x) \supset B(x)))$

By Line 53 we get

77. $(\forall(x)\Phi(x)) \supset (\forall(x)B(x))$ From 68 and 76,

but we also get

78. $B(x) \supset (x \triangleleft x \supset P(x))$

so we have the following:

79. $B(x) \supset P(x)$
80. $(\forall(x)B(x)) \supset (\forall(x) P(x))$
81. $(\forall(x)\Phi(x)) \supset (\forall(x) P(x))$ From 77 and 80
82. $(\forall(x) (\forall(z) (z \triangleleft x \supset P(z)) \supset P(x))) \supset (\forall(x)P(x))$ Rule 6

Other induction rules may then be derived as corollaries as follows.

83. $\sim atom(x) \supset d(x) \triangleleft x$
84. $(\forall(z)(z \triangleleft x \supset P(z))) \supset (d(x) \triangleleft x \supset P(d(x)))$
85. $\sim atom(x) \supset ((\forall(z)(z \triangleleft x \supset P(z))) \supset P(d(x)))$

$$86. \sim atom(x) \supset ((P(d(x)) \supset P(x)) \supset \phi(x))$$

$$87. (P(d(x)) \supset P(x)) \supset (\sim atom(x) \supset \phi(x))$$

$$88. (\forall(x) (P(d(x)) \supset P(x))) \supset (\sim atom(x) \supset \phi(x))$$

$$89. P(x) \supset \phi(x)$$

$$90. (atom(x) \supset P(x)) \supset (atom(x) \supset \phi(x))$$

$$91. (\forall(x) (atom(x) \supset P(x))) \supset (atom(x) \supset \phi(x))$$

$$92. ((\forall(x) (atom(x) \supset P(x))) \wedge (\forall(x) (P(d(x)) \supset P(x)))) \supset \phi(x)$$

From 88 and 91

$$93. ((\forall(x) (atom(x) \supset P(x))) \wedge (\forall(x) (P(d(x)) \supset P(x)))) \supset (\forall(x) \phi(x))$$

$$94. (\forall(x) \phi(x)) \supset (\forall(x) P(x))$$

From 81

$$95. ((\forall(x) (atom(x) \supset P(x))) \wedge (\forall(x) (P(d(x)) \supset P(x)))) \supset (\forall(x) P(x))$$

(Same proof works with d replaced by $addadd$, for example)

4.10.5. Example 5. Generation of Some Labeled Functions,
e.g., maplist.

This may not be the most efficient generation of these functions.

$$nestp^{\wedge}(x) ::= andlistlistcar((\lambda(y,x)(\sim atom(x) \wedge y = d(x))),d(x),x)$$

Easily generatable

$$1. nestp^{\wedge}(x) \equiv nestp^{\wedge}(x)$$

Axiom 6

$$2. nestp^{\wedge}(x) \equiv (d(x) = \phi \vee (\sim atom(d(x)) \wedge \sim atom(x) \wedge \sim atom(a(x)) \wedge ad(x)=da(x) \\ \wedge andlistlistcar((\lambda(y,x) (\sim atom(x) \wedge y = d(x))),dd(x),d(x))))$$

Rules 6 & 7 several times

$$3. \quad \text{nestp}'(x) \equiv (d(x) = \emptyset \vee (\sim \text{atom}(d(x)) \wedge \sim \text{atom}(x) \wedge \sim \text{atom}(a(x)) \\ \wedge ad(x) = da(x) \wedge \text{nestp}'(d(x))))$$

Rule 6

$$\text{nestp}(x) \equiv d(x) = \emptyset \vee (\sim \text{atom}(x) \wedge \sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \\ \wedge ad(x) = da(x) \wedge \text{nestp}(d(x)))$$

Now apply Rule 16 to Line 3 with

$$\begin{array}{l} z := \text{nestp}' \\ u := (x) \\ v := (d(x) = \emptyset \vee (\sim \text{atom}(d(x)) \wedge \sim \text{atom}(x) \wedge ad(x) = da(x) \wedge \text{nestp}'(d(x)))) \end{array}$$

to get

$$4. \quad \textcircled{E} \supset \text{nestp}(x)$$

$$5. \quad \text{orlistcar}(P, x) \equiv \text{orlistcar}(P, x)$$

$$6. \quad \text{orlistcar}(P, x) \equiv \sim \text{andlistcar}((\lambda(x) \sim P(x)), x)$$

$$7. \quad \text{orlistcar}(P, x) \equiv \sim(\text{atom}(x) \vee (\sim P(a(x)) \wedge \text{andlistcar}((\lambda(x) \sim P(x)), d(x))))$$

Rule 7 once & 6 twice

$$8. \quad \text{orlistcar}(P, x) \equiv (\sim \text{atom}(x) \wedge (P(a(x)) \vee \sim \text{andlistcar}((\lambda(x) \sim P(x)), d(x))))$$

By P.C.

$$9. \quad \text{orlistcar}(P, x) \equiv (\sim \text{atom}(x) \wedge (P(a(x)) \vee \text{orlistcar}(P, d(x))))$$

Rule 6

$$10. \quad x \in y \equiv \text{orlistcar}((\lambda(u) u = x), y) \quad \text{Rule 6}$$

$$11. \quad x \in y \equiv (\sim \text{atom}(y) \wedge (a(y) = x \vee \text{orlistcar}((\lambda(u) u = x), d(y))))$$

Line 9 and Rule 6

$$12. \quad x \in y \equiv (\sim \text{atom}(y) \wedge (a(y) = x \vee x \in d(y)))$$

Rule 6

$$\text{nestpp}'(x) \equiv \text{nestp}(x) \wedge \exists(y) (\text{atom}(y) \wedge y \in x)$$

(generate this via Rule 6)

$$13. \text{nestpp}'(x) \equiv (d(x) = \emptyset \vee (\sim \text{atom}(x) \wedge \sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \wedge ad(x) = da(x) \wedge \text{nestp}(d(x)))) \wedge \exists(y) (\text{atom}(y) \wedge (\sim \text{atom}(x) \wedge (a(x) = y \vee y \in d(x))))$$

Rules 7 & 6 several times

$$14. \exists(y) (\text{atom}(y) \wedge (\sim \text{atom}(x) \wedge (a(x) = y \vee x \in d(y)))) \equiv (\sim \text{atom}(x) \wedge (\exists(y) (\text{atom}(y) \wedge a(x) = y) \vee \exists(y) (\text{atom}(y) \wedge y \in d(x))))$$

By P.C.

$$15. \exists(y) (\text{atom}(y) \wedge a(x) = y) \equiv \text{atom}(a(x))$$

By P.C.

$$16. \text{nestpp}'(x) \equiv (d(x) = \emptyset \vee (\sim \text{atom}(x) \wedge \sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \wedge ad(x) = da(x) \wedge \text{nestp}(d(x)))) \wedge \sim \text{atom}(x) \wedge (\text{atom}(a(x)) \vee \exists(y) (\text{atom}(y) \wedge y \in d(x)))$$

From Lines 13, 14 and 15 by P.C.

$$17. y \in d(x) \supset \sim \text{atom}(d(x))$$

From Line 12

$$18. \text{nestpp}'(x) \equiv (\sim \text{atom}(x) \wedge ((\text{atom}(a(x)) \wedge d(x) = \emptyset) \vee (\sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \wedge ad(x) = da(x) \wedge \text{nestp}(d(x)) \wedge \exists(y) (\text{atom}(y) \wedge y \in d(x)))))$$

From Lines 16 and 17 by P.C.

$$19. \text{nestpp}'(x) \equiv (\sim \text{atom}(x) \wedge ((\text{atom}(a(x)) \wedge d(x) = \emptyset) \vee (\sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \wedge ad(x) = da(x) \wedge \text{nestpp}'(d(x)))))$$

By Rule 6

$$\text{nestpp}(x) =: \sim \text{atom}(x) \wedge ((\text{atom}(a(x)) \wedge d(x) = \emptyset) \vee (\sim \text{atom}(d(x)) \wedge \sim \text{atom}(a(x)) \wedge ad(x) = da(x) \wedge \text{nestpp}(d(x))))$$

$$20. \textcircled{P} \supset \text{nestpp}(x)$$

By Rule 16

$$\text{nestpred}(x,y) =: a(y) = x \wedge \text{nestpp}(y)$$

21. $nestpred(x,y) \equiv nestpred(x,y)$ By P.C.
22. $nestpred(x,y) \equiv (a(y) = x \wedge \sim atom(y) \wedge ((atom(a(y)) \wedge d(y) = \emptyset) \vee$
 $(\sim atom(d(y)) \wedge \sim atom(a(y)) \wedge ad(y) = da(y) \wedge nestpp(d(y))))))$
Rules 7 and 6 and P.C.
23. $nestpred(x,y) \equiv (a(y) = x \wedge \sim atom(y) \wedge ((atom(x) \wedge d(y) = \emptyset) \vee$
 $(\sim atom(d(y)) \wedge \sim atom(x) \wedge ad(y) = d(x) \wedge nestpp(d(y))))))$
Via Axiom 7
24. $nestpred(x,y) \equiv (\sim atom(y) \wedge a(y) = x \wedge$
 $((atom(x) \wedge d(y) = \emptyset) \vee (\sim atom(x) \wedge \sim atom(d(y))$
 $\wedge nestpred(d(x),d(y))))))$
Rule 6
25. $nestpred(x,y) \supset \sim atom(y)$ By P. C.
26. $nestpred(d(x),d(y)) \supset \sim atom(d(y))$ Rule 4 and 5
27. $(a(y) = x \wedge d(y) = \emptyset) \equiv a(y) * d(y) = x * \emptyset$ Property of =
28. $\sim atom(y) \equiv y = a(y) * d(y)$ Basic theorem for a & d
29. $(\sim atom(y) \wedge a(y) = x \wedge d(y) = \emptyset) \equiv y = x * \emptyset$
From Lines 27 and 28 by P.C.
30. $nestpred(x,y) \equiv ((atom(x) \wedge y = x * \emptyset) \vee$
 $(\sim atom(x) \wedge \sim atom(y) \wedge x = a(y) \wedge nestpred(d(x),d(y))))$
From Lines 24, 26, and 29 by P.C.
- $nest'(x) =: \iota(y) nestpred(x,y)$
31. $\exists!(y) nestpred(x,y) \supset nestpred(x, \iota(y) nestpred(x,y))$
Substitution into Axiom 14, plus Rule 6
32. $\exists!(y) nestpred(x,y) \supset nestpred(x, nest'(x))$
Rule 6

Thus $nest'$ has been generated.

33. $nestpred(x, x*y) \equiv ((atom(x) \wedge x*y = x * \emptyset) \vee$
 $(\sim atom(x) \wedge \sim atom(x*y) \wedge x = a(x*y) \wedge nestpred(d(x), d(x*y))))$
From Line 30 by Rule 4
34. $(\sim atom(x) \wedge nestpred(d(x), y)) \supset nestpred(x, x*y)$
By P.C. & properties of * and a .
35. $nestpred(x, x*y) \supset \exists(y) nestpred(x, y)$ By P.C.
36. $(\sim atom(x) \wedge nestpred(d(x), y)) \supset \exists(y) nestpred(x, y)$
From 34 and 35 by P.C.
37. $\sim atom(x) \supset (\exists(y) nestpred(d(x), y) \supset \exists(y) nestpred(x, y))$
By P.C.
38. $atom(x) \supset d(x) = x$ Already proved
39. $atom(x) \supset (\exists(y) nestpred(d(x), y) \supset \exists(y) nestpred(x, y))$
Via Axiom 7
40. $\forall(x) (\exists(y) nestpred(d(x), y) \supset \exists(y) nestpred(x, y))$
From 37 and 39 by P.C.
41. $atom(x) \supset nestpred(x, x * \emptyset)$ Put \emptyset for y in Line 33 and use P.C.
42. $\forall(x) (atom(x) \supset \exists(y) nestpred(x, y))$ By P.C.
43. $\forall(x) \exists(y) nestpred(x, y)$ From 40 and 42 via substitution
into induction rule corollary, Section 4.10.4
44. $\exists(y) nestpred(x, y)$ By P.C.
45. $(\sim atom(x) \wedge nestpred(x, y)) \supset (nestpred(d(x), d(y)) \wedge x = a(y))$
From Line 30 via P.C.
46. $(\sim atom(x) \wedge nestpred(x, y) \wedge nestpred(x, z)) \supset$
 $(nestpred(d(x), d(y)) \wedge nestpred(d(x), d(z)) \wedge a(y) = a(z))$
By P.C. and properties of =

$$47. \quad \forall(y,z) ((\text{nestpred}(d(x),y) \wedge \text{nestpred}(d(x),z)) \supset y = z) \supset \\ ((\text{nestpred}(d(x),d(y)) \wedge \text{nestpred}(d(x),d(z))) \supset d(y) = d(z))$$

By P.C.

$$48. \quad (\sim \text{atom}(x) \wedge \forall(y,z) ((\text{nestpred}(d(x),y) \wedge \text{nestpred}(d(x),z)) \supset y = z)) \supset \\ ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset (a(y) = a(z) \wedge d(y) = d(z)))$$

From 46 and 47 by P.C.

$$49. \quad \sim \text{atom}(x) \supset (\forall(y,z) ((\text{nestpred}(d(x),y) \wedge \text{nestpred}(d(x),z)) \supset y = z) \supset \\ \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z))$$

By P.C. and properties of =

$$50. \quad \text{atom}(x) \supset d(x) = x \quad \text{Already proved}$$

$$51. \quad \text{atom}(x) \supset (\forall(y,z) ((\text{nestpred}(d(x),y) \wedge \text{nestpred}(d(x),z)) \supset y = z) \supset \\ \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z))$$

Properties of =

$$52. \quad \forall(x) (\forall(y,z) ((\text{nestpred}(d(x),y) \wedge \text{nestpred}(d(x),z)) \supset y = z) \supset \\ \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z))$$

From 49 and 51 by P.C.

$$53. \quad (\text{atom}(x) \wedge \text{nestpred}(x,y)) \supset y = x * \emptyset$$

From Line 30 via P.C.

$$54. \quad \text{atom}(x) \supset ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z)$$

By P.C. & properties of =

$$55. \quad \forall(x) (\text{atom}(x) \supset \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z))$$

By P.C.

$$56. \quad \forall(x) \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z)$$

From 52 and 55 via substitution
into induction rule corollary, Section 4.10.4.

$$57. \quad \forall(y,z) ((\text{nestpred}(x,y) \wedge \text{nestpred}(x,z)) \supset y = z)$$

By P.C.

58. $\exists!(y)nestpred(x,y)$ From 44 and 57 via substitution into Axiom 13
59. $\exists!(y)nestpred(x,y) \supset nestpred(x, nest'(x))$
Repeat of Line 32
60. $nestpred(x, nest'(x))$ Modus ponens
61. $(nestpred(x,y) \wedge nestpred(x,z)) \supset y = z$ From 57 by P.C.
62. $(nestpred(x, nest'(x)) \wedge nestpred(x,y)) \supset y = nest'(x)$ Rules 4 and 5
63. $nestpred(x, nest'(x))$ Repeat of Line 60
64. $nestpred(x,y) \supset y = nest'(x)$ From 62 and 63 by P.C.
65. $y = nest'(x) \supset (nestpred(x, nest'(x)) \supset nestpred(x,y))$ Axiom 7
66. $y = nest'(x) \supset nestpred(x,y)$ From 63 and 65 by P.C.
67. $nestpred(x,y) \equiv y = nest'(x)$ From 64 and 66
68. $(atom(x) \wedge nest'(x) = x * \emptyset) \vee (\sim atom(x) \wedge \sim atom(nest'(x)) \wedge x = a(nest'(x)) \wedge nestpred(d(x), d(nest'(x))))$
From Line 30 and 60
69. $\sim atom(x) \supset (\sim atom(nest'(x)) \wedge x = a(nest'(x)) \wedge nestpred(d(x), d(nest'(x))))$
By P.C.
70. $nestpred(d(x), d(nest'(x))) \equiv d(nest'(x)) = nest'(d(x))$
Substitution into Line 67
71. $\sim atom(nest'(x)) \equiv nest'(x) = a(nest'(x)) * d(nest'(x))$
Properties of a and d
72. $\sim atom(x) \supset (nest'(x) = a(nest'(x)) * d(nest'(x)) \wedge a(nest'(x)) = x \wedge d(nest'(x)) = nest'(d(x)))$
From 69, 70, and 71 by P.C.
73. $\sim atom(x) \supset nest'(x) = x * nest'(d(x))$
Properties of $=$ and P.C.

$$74. (\sim g(x) \supset f(x) = x * f(d(x))) \equiv (\sim g(x) \supset f(x) = [\oplus \rightarrow x * f(d(x))])$$

Via Rule 11 with $z =: x * f(d(x))$

$$75. (\sim atom(x) \supset nest'(x) = x * nest'(d(x))) \equiv (\sim atom(x) \supset nest'(x) = [\oplus \rightarrow x * nest'(d(x))])$$

Rule 5

$$76. \sim atom(x) \supset nest'(x) = [\oplus \rightarrow x * nest'(d(x))]$$

From 73 and 75 by P.C.

$$77. atom(x) \supset nest'(x) = x * \emptyset$$

From 68 by P.C.

$$78. f(x) = [g(x) \rightarrow x * \emptyset; \oplus \rightarrow x * f(d(x))] \equiv$$

$$((g(x) \supset f(x) = x * \emptyset) \wedge (\sim g(x) \supset f(x) = [\oplus \rightarrow x * f(d(x))]))$$

Rule 9 with $z =: ([g(x) \rightarrow x * \emptyset; \oplus \rightarrow x * f(d(x))])$

$$79. nest'(x) = [atom(x) \rightarrow x * \emptyset; \oplus \rightarrow x * nest'(d(x))] \equiv$$

$$((atom(x) \supset nest'(x) = x * \emptyset) \wedge$$

$$(\sim atom(x) \supset nest'(x) = [\oplus \rightarrow x * nest'(d(x))]))$$

Rule 5

$$80. nest'(x) = [atom(x) \rightarrow x * \emptyset; \oplus \rightarrow x * nest'(d(x))]$$

From 76, 77, and 79 by P.C.

$$nest(x) =: [atom(x) \rightarrow x * \emptyset; \oplus \rightarrow x * nest(d(x))]$$

$$81. \mathfrak{H}(z) \ z = nest(x)$$

From 80 by Rule 17 with

$$z =: nest'$$

$$u =: (x)$$

$$v =: [atom(x) \rightarrow x * \emptyset; \oplus \rightarrow x * nest'(d(x))]$$

So *nest* has been generated!

$$\text{maplist}'(f,x) =: \text{maplistcar}(f,\text{nest}(x))$$

$$82. \text{maplist}'(f,x) = [\text{atom}(x) \rightarrow x; \oplus \rightarrow f(a(\text{nest}(x))) * \text{maplistcar}(f,d(\text{nest}(x)))]$$

From Axiom 6 and Rules 4, 5, 7, & 6.

$$83. (\text{atom}(x) \supset \text{maplist}'(f,x) = x) \wedge$$

$$(\sim \text{atom}(x) \supset \text{maplist}'(f,x) = [\oplus \rightarrow f(a(\text{nest}(x))) * \text{maplistcar}(f,d(\text{nest}(x)))])$$

Via Rule 9 as before

$$84. \text{nest}(x) = [\text{atom}(x) \rightarrow x * \emptyset; \oplus \rightarrow x * \text{nest}(d(x))]$$

From Axiom 6 via rules 4,5,7, & 6.

$$85. (\text{atom}(x) \supset \text{nest}(x) = x * \emptyset) \wedge (\sim \text{atom}(x) \supset \text{nest}(x) = [\oplus \rightarrow x * \text{nest}(d(x))])$$

Via Rule 9 as before

$$86. \sim \text{atom}(x) \supset \text{nest}(x) = [\oplus \rightarrow x * \text{nest}(d(x))]$$

By P.C.

$$87. \sim \text{atom}(x) \supset \text{nest}(x) = x * \text{nest}(d(x))$$

Via Rule 11 as before

$$88. \sim \text{atom}(x) \supset (a(\text{nest}(x)) = x \wedge d(\text{nest}(x)) = \text{nest}(d(x)))$$

Properties of a , d , and $=$.

$$89. (\text{atom}(x) \supset \text{maplist}'(f,x) = x) \wedge (\sim \text{atom}(x) \supset$$

$$\text{maplist}'(f,x) = [\oplus \rightarrow f(x) * \text{maplistcar}(f,\text{nest}(d(x)))]$$

From 83, 88, and Axiom 7 by P.C.

$$90. (\text{atom}(x) \supset \text{maplist}'(f,x) = x) \wedge (\sim \text{atom}(x) \supset$$

$$\text{maplist}'(f,x) = [\oplus \rightarrow f(x) * \text{maplist}(f,d(x))]$$

Rule 6

$$91. \text{maplist}'(f,x) = [\text{atom}(x) \rightarrow x; \oplus \rightarrow f(x) * \text{maplist}'(f,d(x))]$$

Via Rule 9 as before

$$\text{maplist}(f,x) =: [\text{atom}(x) \rightarrow x; \oplus \rightarrow f(x) * \text{maplist}(f,d(x))]$$

$$92. \exists(z) z = \text{maplist}(f,x)$$

Via Rule 17 as before

If we wished we could go on to prove inductively such theorems as

$$\text{maplist}(f,x) = \text{maplist}'(f,x)$$

and

$$\text{maplistcar}(f,x) = \text{maplist}((\lambda(x) f(a(x))), x)$$

we could similarly define

$$\text{andlist}'(P,x) = \text{andlistcar}(P,\text{nest}(x))$$

$$\text{andlist}(P,x) = [\text{atom}(x) \rightarrow \oplus ; \oplus \rightarrow P(x) \wedge \text{andlist}(P,d(x))]$$

and prove theorems

$$\text{andlist}(P,x) \equiv \text{andlist}'(P,x)$$

and

$$\text{andlistcar}(P,x) \equiv \text{andlist}((\lambda(x) P(a(x))), x)$$

4.10.6 Example 6: The Predecessor Function

Suppose we have \boxed{S} defined as in Section 2.1.1.9, and suppose we have as theorems the Peano axioms for \boxed{S} , i.e.

- A $\sim(\emptyset = \boxed{S}(x))$
- B $\boxed{S}(x) = \boxed{S}(y) \supset x = y$
- C $(P(\emptyset) \wedge \forall(x) (P(x) \supset P(\boxed{S}(x)))) \supset \forall(x) P(x)$

(Note: This does not give us enough power to prove the true statements of form $\boxed{S}(\alpha) = \beta$. To prove these we would have to restore the ordering of the atoms which we threw away in Section 2.1.1.10. One way to do this would be to add back into the system the rules in the footnote near the end of Section 2.1.1.9.)

We now consider the predecessor function whose ordinary name is

predecessor(x) =: $\iota(y) ((x = \emptyset \wedge y = x) \vee \boxed{S}(y) = x)$. We first want to generate an algorithmic name of this form:

$(\lambda, (x), ((label, f, (\lambda(z, x) [\boxed{S}(z) = x+z; x = \emptyset \rightarrow x; \emptyset \rightarrow f(\boxed{S}(z), x)])), \emptyset, x))$

We use the dummy z to count up from zero to the predecessor.

Our first job is to introduce the z. This is easy. Define

dummypred(z, x) =: *predecessor*(x). Now *dummypred* is a complete function of z and x. We easily prove $\exists!(y) ((x = \emptyset \wedge y = x) \vee \boxed{S}(y) = x)$ so by axiom 14 we get

1. $(x = \emptyset \wedge \text{dummypred}(z, x) = x) \vee \boxed{S}(\text{dummypred}(z, x)) = x$. We then proceed as follows:

2. *dummypred*(z, x) = *predecessor*(x)

3. *dummypred*(z, x) = *dummypred*($\boxed{S}(z)$, x)

4. $\sim x = \emptyset \supset \text{dummypred}(z, x) = [\emptyset \rightarrow \text{dummypred}(\boxed{S}(z), x)]$ by Rule 11

5. $x = \emptyset \supset \text{dummypred}(z, x) = x$ from line 1 and A above

6. *dummypred*(z, x) = $[x = \emptyset \rightarrow x; \emptyset \rightarrow \text{dummypred}(\boxed{S}(z), x)]$ from 4 and 5

by Rule 9

7. $\sim \boxed{S}(z) = x \supset \text{dummypred}(z, x) = [x = \emptyset \rightarrow x; \oplus \rightarrow \text{dummypred}(\boxed{S}(z), z)]$
8. $\boxed{S}(z) = x \supset \boxed{S}(\text{dummypred}(z, x)) = x$ from lines 1 and A above
9. $\boxed{S}(z) = x \supset \text{dummypred}(z, x) = z$ from Lines 8 and B
10. $\text{dummypred}(z, x) = [\boxed{S}(z) = x \rightarrow z; x = \emptyset \rightarrow x; \oplus \rightarrow \text{dummypred}(\boxed{S}(z), x)]$

Hence we can generate the algorithmic name

$$\text{dummyalg}(z, x) = [\boxed{S}(z) = x \rightarrow z; x = \emptyset \rightarrow x; \oplus \rightarrow \text{dummypred}(\boxed{S}(z), x)]$$

via Rule 17.

Thus from the ordinary name, *dummypred* we generate the algorithmic name *dummyalg*.

Notice, however, that *dummypred* and *dummyalg* name different functions. *dummypred* names a complete function, and *dummyalg* does not, the function named by *dummyalg* being undefined for $\emptyset < x \leq z$, This algorithmic name, then, contains less information than the ordinary name from which it is generated. This is not unusual. Of course an algorithmic name cannot contain more information than the name from which it was generated. An ordinary name for the function named by *dummyalg* might be given by $\text{dummyalg}(z, x) =$

$i(y) ((x = \emptyset \wedge y = x) \vee (z \leq x \wedge \boxed{S}(y) = x))$ if we had defined \leq which we have not. Thus it is not immediately obvious how to generate the algorithmic name *dummyalg* from an ordinary name like the one indicated here. Might it be the case that there exists an algorithmic name for a partial function which cannot be generated from any ordinary name of that function, but which can be generated from an ordinary name of some more complete function? I do not know the answer to this question.

Now define $\text{predalg}(x) =: \text{dummyalg}(\emptyset, x)$. Note that although *dummyalg* is not a complete function, *predalg* is a complete function.

Of course if \boxed{S} is constructed as in Section 2.1.1.9, we have a much simpler way to generate *predalg*. In Section 2.1.1.9

$\boxed{S}(x) = \boxed{gn^{-1}}(\boxed{p} * \boxed{gn}(x))$ holds, where \boxed{gn} is like gn and $\boxed{gn^{-1}}$ is like gn^{-1} except that the substitutions of (qu, p) for \tilde{p} , $(qu, \hat{1})$ for $\tilde{\hat{1}}$, and $(qu, \hat{2})$ for $\tilde{\hat{2}}$ have been made. Hence \boxed{gn} and $\boxed{gn^{-1}}$ are already generated algorithmic names and we can define $predalg(x) = \boxed{gn^{-1}}(d(\boxed{gn}(x)))$. We can also define $x \boxed{\leq} y =: \boxed{gn}(x) \leq \boxed{gn}(y)$.

4.10.7 Example 7: The μ Schema

Here we shall sketch the method we use to generate an algorithmic name for a function defined by the μ schema of recursive function theory. In our notation, such a definition would be of the form

$\phi(x) =: \iota(y) (\Pi(y, x) \wedge \forall(v) (\Pi(v, x) \supset v = y))$. Again we introduce a dummy z to get the algorithmic name

$(\lambda, (x), ((label, f, (\lambda(z, x) [\Pi(z, x) \rightarrow z; \Theta \rightarrow f(\boxed{S}(z), x)])), \emptyset, x))$.

We are tempted to define the partial function *dummyphi*(z, x) =:

$\iota(y)(z \boxed{\leq} y \wedge \Pi(y, x) \wedge \forall(v) ((z \boxed{\leq} v \wedge \Pi(v, x)) \supset y \boxed{\leq} v))$, and generate an algorithm for it. But we first need $\sim \Pi(z, x) \supset dummyphi(z, x) = dummyphi(\boxed{S}(z), x)$

which does not necessarily hold when the z and x range outside the domain of definition of the function. The closest we can come to generating our algorithmic name is

$\exists(w) (z \boxed{\leq} w \wedge \Pi(w, x)) \supset dummyphi(z, x) = [\Pi(z, x) \rightarrow z; \Theta \rightarrow dummyphi(\boxed{S}(z), x)]$.

If only we could prove

A. $\sim \exists(w) (z \boxed{\leq} w \wedge \Pi(w, x)) \supset dummyphi(z, x) = dummyphi(\boxed{S}(z), x)$ we could prove $dummyphi(z, x) = [\Pi(z, x) \rightarrow z; \Theta \rightarrow dummyphi(\boxed{S}(z), x)]$. Then we could generate

$dummyphialg(z, x) =: [\Pi(z, x) \rightarrow z; \Theta \rightarrow dummyphialg(\boxed{S}(z), x)]$, and we

could define $phialg(x) =: dummyphialg(\emptyset, x)$ as we wished. But we cannot prove line A. It does not necessarily hold.

However, if we artificially complete the partial function *dummyphi* we can do it. Instead of *dummyphi*, use *dummyphicomplete*, defined by

$$\text{dummyphicomplete}(z, x) =: \neg(y) ((\neg H(w) (z \sqsubseteq w \wedge \Pi(w, x)) \wedge y = \emptyset) \\ \vee (H(w) (z \sqsubseteq w \wedge \Pi(w, x)) \wedge (z \sqsubseteq y \wedge \Pi(y, x) \wedge \forall(v)((z \sqsubseteq v \wedge \Pi(v, x)) \supset v=y))))$$

, in the preceding discussion and then the generation works. Once again, although *dummyphialg* gives an algorithm for the function *dummyphi*, we found it most difficult to generate *dummyphialg* from *dummyphi* and found it more convenient to generate it from *dummyphicomplete* which is quite a different function. Notice also: we cannot prove

$\text{dummyphi}(z, x) = \text{dummyphialg}(z, x)$ since we have no information on the value of *dummyphi* when $\neg H(w) (z \sqsubseteq w \wedge \Pi(w, x))$ holds. To prove that two function names name the same function, it is not sufficient to prove that they give the same values over their domains of definition. If the functions are partial, the names may be regarded as incomplete descriptions of complete functions (not necessarily recursive). In that case, the two descriptions must give enough information for us to say that any two complete functions that might be described by the two descriptions must give identical values everywhere, though we might not be able to say what those values are. For example, let us define

$$\text{newdummyphialg}(z, x) =: [H(w) (z \sqsubseteq w \wedge \Pi(w, x)) \rightarrow \text{dummyphialg}(z, x) ; \emptyset \rightarrow \text{dummyphi}(z, x)] .$$

Then we can prove

$\text{dummyphi}(z, x) = \text{newdummyphialg}(z, x)$ even though both functions are partial, i.e., both descriptions are incomplete.

4.10.8 Example 8: $T(x) \supset F(x)$

We shall give only a very sketchy outline of the proof of $T(x) \supset F(x)$. We can easily prove $\text{Proof}(x) \supset (y \in x \supset T(y))$.

We now induct on the length of proof as follows: Temporarily define

$$\Pi(x) =: y \in x \supset F(y).$$

Then we can prove the following sequence.

1. $(\Pi(d(x) \wedge Pfstep(x)) \supset \Pi(x))$ by induction on expression named by x
2. $Axiom(x) \supset F(x)$
3. $(\Pi(d(x) \wedge axiom(a(x))) \supset \Pi(x))$ from 2.
4. $((Proof(d(x) \supset \Pi(d(x))) \wedge Proof(x)) \supset \Pi(x))$ from 1 and 3.
5. $Proof(x) \supset \Pi(x)$ from 4 by induction theorem of Example 4.
6. $T(x) \supset F(x)$ from 5.

4.10.9 Example 9: Sketch of Proof of $T(x) \supset T(apl(x))$

We shall here give an outline of how one would prove $T(x) \supset T(apl(x))$. We can give no more than the barest outline because the proof is extremely long and tedious.

We begin by developing alternative formulations of some rules of inference.

The mechanism we have in Rule 3 for handling bound variables is very inefficient. We need a predicate which says that two expressions are identical except for alphabetic differences in bound variables. Such a predicate is \equiv (Table 9). We will need a whole corpus of theorems incorporating this which will allow us to automatically make alphabetic changes of bound variables as needed. We shall not develop these theorems here.

By way of example we have the following theorem which is provable from Rule 3 and equivalent to Rule 3. Alternative Rule 3:

$$(T(x) \wedge x \equiv y) \supset T(y)$$

The proof is rather tedious. Note also: \equiv is an equivalence relation.

The form of Rules 4-7 is inconvenient for some purpose because in order that *freecheck* hold where required we must, in general, do some preliminary juggling of bound variables via Rule 3. We would like a recursive, legal, way to do the juggling. To this end we have defined *freefix* and *freefixlists* in Table 9. Their most important properties are summarized by the theorems

below. (The proofs are rather tedious.)

$$z = \text{freefix}(x, y, z)$$

$$(x = y \wedge \text{Snf}(x, y, z) = \text{Snf}(x, y, w)) \supset z = w$$

$$\text{freecheck}(x, y, \text{freefix}(x, y, z))$$

$$z = \text{freefixlists}(x, y, z)$$

$$\text{andlista}((\lambda(u) \text{freecheck}(u, y, \text{freefixlists}(x, y, z))), x)$$

So that we can also prove

$$\text{andlista}((\lambda(v) \sim (\text{listbinder } (v) \vee v = \text{label})), y) \supset$$

$$\text{andlistlista}((\lambda(u, v) \text{freecheck}(u, v, \text{freefixlists}(x, y, z))), x, y) .$$

Write β for $\left(\left(aa(y) \right) \left(ada(y) \right) \left(\text{freefixlists}(ada(y), d(y), \text{adda}(y)) \right) \right) * d(y)$ so

$$y = \beta \text{ and } u = \text{Snf}(y, \beta, u) \text{ and } v = \text{Snf}(y, \beta, v) \text{ hold. Then we can make}$$

the following substitutions into the theorem which is Rule 6.

for y put β

for u put $\text{Snf}(y, \beta, u)$

for v put $\text{Snf}(y, \beta, v)$

This gives, after some manipulation and replacement by equivalences,

$$(T (\text{Snf}(y, \beta, u)) \wedge \text{expression}(\text{Snf}(y, \beta, v)) \wedge aa(y) = \lambda)$$

$$\wedge \text{andlistlista}((\lambda(x, z) \text{freecheck}(x, z,$$

$$\text{freefixlists}(ada(y), d(y), \text{adda}(y))), ada(y), d(y))$$

$$\wedge \text{Snf}(\beta, \text{Ssuffixl}(ada(y), d(y), \text{adda}(y)), \text{Snf}(y, \beta, u)) =$$

$$\text{Snf}(\beta, \text{Ssuffixl}(ada(y), d(y), \text{adda}(y)), \text{Snf}(y, \beta, v))$$

$$\wedge (\beta \leq \text{Snf}(y, \beta, u) \vee (\text{andlista}((\lambda(z) \sim z \leq \text{adda}(\beta)), \text{ada}(\beta)))$$

$$\wedge \text{andlistlista}((\lambda(x, z) \text{label} \leq z \supset x \leq \text{adda}(\beta)), \text{ada}(\beta), d(\beta))))$$

$$\supset T(\text{Snf}(y, \beta, v))$$

Using this, we can prove

alternative Rule 6:

$$\begin{aligned}
 & (T(u) \wedge \text{expression}(v) \wedge aa(y) = \lambda) \\
 & \wedge \text{Snf}(y, \text{Ssffixl}(ada(y), d(y), adda(y)), u) = \\
 & \quad \text{Snf}(y, \text{Ssffixl}(ada(y), d(y), adda(y)), v) \\
 & \wedge \text{newvarex}(y, u) \leq \text{Snf}(y, \text{newvarex}(y, u), u) \supset \\
 & T(v)
 \end{aligned}$$

Likewise we can prove

alternative Rule 7:

$$\begin{aligned}
 & (T(u) \wedge \text{expression}(v) \wedge a(y) = \text{label} \wedge ad(y) \leq add(y) \\
 & \wedge \text{Snf}(y, \text{Sffix}(ad(y), y, add(y)), u) = \\
 & \quad \text{Snf}(y, \text{Sffix}(ad(y), y, add(y)), v) \supset \\
 & T(v)
 \end{aligned}$$

alternative Rule 4:

$$\begin{aligned}
 & (T(y) \wedge \text{variable}(x) \wedge \text{Simplexpr}(z) \wedge \text{type}(x) = \text{expertype}(z)) \supset \\
 & T(\text{Sffix}(x, z, y))
 \end{aligned}$$

alternative Rule 5:

$$\begin{aligned}
 & (T(y) \wedge T(u) \wedge z \leq u \wedge (\text{Pfv}(x) \vee \text{Ifv}(x)) \wedge \text{type}(x) = \text{expertype}(z)) \supset \\
 & T(\text{Sffix}(x, z, y))
 \end{aligned}$$

In these formulations, the bound variables take care of themselves automatically. Except for the alternative Rule 6, the alternative rules are as powerful as the original rules.

From the alternative formulations of Rules 6 and 7 used as theorems, together with other rules, we can prove

$$(F(x) \supset T(x \oplus \text{oneapl}(x))) \wedge (Tm(x) \supset T(x \ominus \text{oneapl}(x)))$$

The proof is by tedious induction on the S-expression named by x. We shall not

reproduce this proof here.

From the above theorem,

$T(x) \supset T(\text{oneapl}(x))$ follows as a special case.

A glance at *apl* convinces us that it is a non-contradictory function expression. It is even the name of a complete function. It may not be obvious, however, just how this function is to be generated. The following function expression names the same function.

$$\text{apl}'(x) =: \lambda(y) (\text{oneapl}(y) = y \wedge \exists(z) (y = a(z) \wedge \text{andlist}((\lambda(v) (v = \emptyset \vee v = (x) \vee a(v) = \text{oneapl}(ad(v))))), z)) .$$

(*andlist* is defined at the end of Section 4.10.5)

However, to prove the necessary recursion theorem on *apl'* to allow us to generate *apl* via Rule 17 is very complicated. It requires a set of list handling functions and a corpus of theorems about them to allow us to prove the required uniqueness of the *z* in the definition of *apl'*. Some of these functions (e.g. *maplist*) and theorems were developed in earlier examples. The methods referred to here have been illustrated on less complicated examples. These functions, theorems, and methods are necessary both to handle proofs and to handle sequences of formulae like the *z* sequence above. One use of these methods, induction on length of proof, was indicated in our sketch of the proof of $T(x) \supset F(x)$ in Section 4.10.8. A similar, but more complicated, induction on the length of the *z* list of the *apl'* definition gives us $T(x) \supset T(\text{apl}(x))$.

4.11 Implementation Routines--Effector Algorithms Used in Section 3

4.11.1 Basic Functions and Notation. These algorithms utilize the following basic functions described in Section 3.

1. Moving functions

derivations

rule

down

antecedents

derivativesusing

variable

2. Explicit net-changing functions

join

constructderivation

constructparameterderivation

3. Node predicates

T-tagged

H-tagged

4. Node modifiers

T-tag

H-tag

5. Search functions

bestlist

bestproductinlist

bestnetrule

bestnetparameterrule

bestnetparametervalue

6. Implicit net-changing functions

erasepunish

punish

7. Hybrid moving function

jumpback

8. Net-changing functional

operate

9. Timed LISP functions

result

requiredantecedents

10. LISP functions

pair

proj₁

proj₂

find

variablein

The notation for the following algorithm definitions is similar to the notation of the "definition" statements in Section 4.3, which, in turn, resembles LISP m-notation. In the following definitions we also employ the LISP PROG notation, together with the operate statements described in Section 3.4.3. In these definitions we also use set theory notation to abbreviate sequences the order of whose members is unimportant. E.g.

$\langle \alpha \cup \{ \beta, \gamma \} \cup \{ y \mid y \in \delta \wedge \text{H-tagged}(y) \} \rangle$ names a sequence whose

members are: (1) the members of the sequence α ; (2) β and γ ;

(3) those members of the sequence δ on which *H-tagged* holds. The dummy

variables in the following algorithm definitions will not be restricted to the variable symbols of Section 2, but may be any Greek or Latin letter. The reader must be careful to not confuse, for example, the bug value ξ with the dummy variable (or bug) ξ which ranges over bug values. It will be clear from context which is meant.

4.11.2 Routines Which Return a Bug Value

```

refineproof( $\xi$ , k) =: prog((  $\Delta$  ) ;
  [ T-tagged( $\xi$ )  $\rightarrow$  return( $\xi$ ) ] ;
   $\Delta := derivations(\xi)$  ;
  return(refineproff( $\xi$  ,  $\Delta$ , k)) )

```

```

refineproff( $\xi$ ,  $\Delta$ , k) =: prog(( $\delta$ ,  $\pi$ ,  $\rho$ ,  $\phi$ ,  $\zeta$  ) ;

```

```

 $\delta := bestlist(\Delta, k)$  ;

```

```

[  $\delta = \emptyset \rightarrow$  return( $\xi$ ) ] ;

```

```

 $\pi := rule(\delta)$  ;

```

```

 $\rho := refineproof(\pi, \uparrow k)$  ;

```

```

 $\phi := operate(\rho)$  ;

```

```

[ T-tagged( $\rho$ )  $\vee$   $\rho =$   $\underbrace{T( ( \underbrace{T( ( \underbrace{qu, \underbrace{x}}_{( )} )}_{( )} ) )}_{( )} \supset T(x) \rightarrow$ 
  return(refinederiv( $\phi(\delta)$  , k)) ] ;

```

```

 $\zeta := expandheuristic(\phi(\delta) , \rho, k)$  ;

```

```

[ T-tagged( $\zeta$ )  $\rightarrow$  return( $\zeta$ ) ] ;

```

```

 $\phi := operate( punish(\xi, down(\delta)) )$  ;

```

```

return( refineproff( $\phi(\xi)$  ,  $\phi(\Delta) - \{\delta\}$  ,  $\uparrow k$ ) )

```

```

refinederiv( $\delta, k$ ) =: refinederivv( $\delta$ , antecedents( $\delta$ ) , k)

```

```

refinederivv( $\delta, \tau, k$ ) =: prog(( $\zeta, \phi$  ) ;
    [ $\tau = \emptyset \rightarrow$  return(T-tag(down( $\delta$ ))) ] ;
 $\zeta :=$  refineproof(a( $\tau$ ),  $k$ ) ;
    [ $\sim$  T-tagged( $\zeta$ )  $\rightarrow$  return(down( $\delta$ )) ] ;
 $\phi =$  operate( $\zeta$ ) ;
    return(refinederiv( $\phi(\delta)$  ,  $\phi(d(\tau))$  ,  $k$ ) )

```

```

expandheuristic( $\delta, \rho, k$ ) =:
    expandheuristicc( $\delta, derivativesusing(\rho) - \{\delta\}, \rho, k$ )

```

```

expandheuristicc( $\delta, \Gamma, \rho, k$ ) =: prog(( $\gamma, \zeta, \phi$  ) ;
 $\gamma :=$  bestlist( $\Gamma, k$ ) ;
    [ $\gamma = \emptyset \rightarrow$  return(down( $\delta$ )) ] ;
 $\zeta :=$  refinebyexample( $\delta, \gamma, \rho, k$ ) ;
    [T-tagged( $\zeta$ )  $\rightarrow$  return( $\zeta$ ) ] ;
 $\phi :=$  operate(punish( $\delta, \gamma$ )) ;
    return(expandheuristicc( $\phi(\delta)$  ,  $\phi(\Gamma - \{\gamma\})$  ,  $\phi(\rho)$  ,  $\uparrow k$ )) )

```

In the definition of *refinebyexample* below, Φ and Θ will stand for functions which are described by the following paragraphs.

If δ , γ , and ξ are bug values with $\boxed{\delta}$ the same as $\boxed{\gamma}$, with $\hat{\delta}$ and $\hat{\gamma}$ derivation nodes, and with $\hat{\xi} = \text{down}(\hat{\delta})$ holding, then $\Phi(\hat{\delta}, \hat{\gamma})$ names a bug value η with these two properties: first, $\boxed{\eta}$ may be formed from $\boxed{\xi}$ by erasure of nodes $\hat{\delta}$ and $\hat{\gamma}$ and removal of all H-tags from $\hat{\xi}$; second, $\hat{\eta}$ is that node referred to above from which the H-tags were removed.

(Remember, we made ad hoc provision for saving all of a net if an erasure splits it in two.)

Suppose now that ξ may be formed from η by addition of certain nodes and change of certain flags. Then we can imagine a net ω which can be formed from ξ by adding back those two nodes that were erased from ξ to make η . Can we unambiguously state where these two nodes should be added back, or are several such ω 's possible? In most cases there is only one ω possible which gives the added nodes the same position relative to other nodes that they had in ξ . When more than one ω is possible one can rely on the fact that all these nets are part of the same patched net to determine the unique "natural" ω , the ω which has the largest number of its nodes "equivalent" to nodes in ξ (where we call a node in ω "equivalent" to a node in ξ if the two nodes are the same node of the patched net). $\hat{\omega}$ and $\hat{\xi}$ are to be the same nodes of the patched net. θ is defined for bug values described as above, so that $\theta(\xi, \eta, \zeta)$ names ω .

```

refinebyexample( $\delta, \gamma, \rho, k$ ) := prog( ( $\xi, x, \phi, \ell, \zeta, \theta$ ) ;
 $\xi := \text{down}(\delta)$  ;
 $x := \text{down}(\gamma)$  ;
 $\phi := \text{operate}(\phi(\delta, \gamma))$  ;
 $\ell := \{ \langle \langle x, \xi \rangle \rangle, \langle \langle \text{ad}(\text{parameter}(\gamma)) \rangle \rangle, \langle \langle \text{ad}(\text{parameter}(\delta)) \rangle \rangle \} \cup$ 
      pair(antecedents( $\gamma$ ), antecedents( $\delta$ )) ;
 $\zeta := \text{prove}(\phi(\xi), \phi(x), \phi(\ell), \phi(\rho), k)$  ;
[ $\sim$ H-tagged( $a(\zeta)$ )  $\rightarrow$  return( $\xi$ )] ;
 $\theta := \text{operate}(\theta(\xi, \eta, \zeta))$  ;
return(refineproof( $\theta(a(\zeta)), \uparrow k$ )) )

```

4.11.3 Routines Which Return a Bug Value Paired With a Sequence of Pairs

```

prove( $\xi$ , x,  $\ell$ ,  $\rho$ , k) =: prog((m,  $\phi$ ,  $\zeta$ , L ) );
  [ H-tagged(  $\xi$  )  $\rightarrow$  return( $\xi$  *  $\ell$  ) ] ;
  [ x =  $\emptyset$   $\rightarrow$  go(A) ] ;
  m := { y | y  $\in$   $\ell$   $\wedge$  ad(y)  $\neq$   $\xi$  } ;
  [ x  $\in$  proj1(m)  $\rightarrow$ 
 $\phi$  := operate(join( $\xi$  , ad(find(x, m)))) ] ;
  [ x  $\in$  proj1(m)  $\wedge$   $\overline{\xi} = \overline{ad(find(x, m))}$   $\rightarrow$ 
    return( prove( $\phi(\xi)$  ,  $\phi(x)$  ,  $\phi(\ell)$   $\phi(\rho)$  , k) ) ] ;

 $\zeta$  := checkprove( $\xi, x, \ell, \rho, k$ ) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] ;
A L := { y | y  $\in$   $\ell$   $\wedge$   $\overline{ad(y)} = \overline{\xi}$  } ;
 $\zeta$  := checklistprove( $\xi, L, \ell, \rho, k$ ) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] ;
  return(randomprove( $\xi, \ell, \rho, k$ )) )

```

$checkprove(\xi, x, \ell, \rho, k) =: checkprovv(\xi, derivations(x), \ell, \rho, k)$

```

checkprovv( $\xi, \Delta, \ell, \rho, k$ ) =: prog((  $\delta, \zeta, \psi$  ) ;
   $\delta$  := bestlist(  $\Delta, k$  ) ;
  [  $\delta = \emptyset$   $\rightarrow$  return (  $\xi$  *  $\ell$  ) ] ;
   $\zeta$  := parametergenerate( $\xi, \delta, \ell, \rho, k$ ) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] ;
   $\psi$  := operate(punish(  $\xi$  , jumpback( $\xi, \delta$  ))) ;
  return(checkprovv( $\psi(\xi)$  ,  $\psi(d(\Delta))$ ,  $\psi(\ell)$ ,  $\psi(\rho)$ ,  $\psi(k)$  ) )

```

```

checklistprove( $\xi$  , L,  $\ell$ ,  $\rho$ , k) =:  prog(( y,  $\phi$ ,  $\zeta$ ,  $\psi$  )
  y := bestproductinlist(L, k) ;
  [ y =  $\emptyset$   $\rightarrow$  return( $\xi$  *  $\ell$ ) ] ;
   $\phi$  := operate(join( $\xi$ , ad(y))) ;
   $\zeta$  := checkprove( $\phi$ ( $\xi$ ) ,  $\phi$ (a(y)) ,  $\phi$ ( $\ell$ ),  $\phi$ ( $\rho$ ), k) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] ;
   $\psi$  := operate( punish( $\xi$ , jumpback( $\xi$ ,a(y)))) ;
  return( checklistprove( $\psi$ ( $\xi$ ),  $\psi$ (d(L)) ,  $\psi$ ( $\ell$ ),  $\psi$ ( $\rho$ ) + k) )

```

In the following definition it is important that $\sigma = \text{bestnetrule}(\sigma, k)$ never hold. *bestnetrule* can be modified so it never does.

```

randomprove(  $\xi$ ,  $\ell$ ,  $\sigma$ , k) =:  prog(( $\rho$ ,  $\zeta$ ,  $\psi$  ) ;
   $\rho$  := bestnetrule( $\sigma$ , k) ;
  [ $\rho = \emptyset \rightarrow$  return ( $\xi$  *  $\ell$ ) ] ;
   $\zeta$  := overallcheck( $\rho$ ,  $\xi$ ,  $\ell$ ,  $\sigma$ , k) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\xi$ ) ] ;
   $\psi$  := operate(punish( $\xi$  , jumpback( $\xi$ , $\rho$ ))) ;
  return (randomprove ( $\psi$ ( $\xi$ ) ,  $\psi$ ( $\ell$ ),  $\psi$ ( $\sigma$ ), + k) )

```

```

parametergenerate( $\xi$ ,  $\delta$ ,  $\ell$ ,  $\sigma$ , k) =:  prog(( $\rho$ , $v$ , $\chi$ , $\omega$ , $\zeta$  ) ;
   $\rho$  := rule ( $\delta$ ) ;
   $v$  := parameter ( $\delta$ ) ;
   $\chi$  := antecedents ( $\delta$ ) ;
  [  $\sim$  ad( $v$ )  $\in$  proj1( $\ell$ )  $\rightarrow$  go (A) ] ;
   $\omega$  :=  $\bigoplus_{\rho} \text{ad}(\text{find}(\text{ad}(v) , \ell ))$  ;
  [ result( $\bar{\rho}$ ,  $\omega$ , k)  $\neq$   $\bar{\xi} \rightarrow$  go (A) ] ;
   $\zeta$  := parametercheckgen( $\rho$ ,  $\omega$ ,  $\xi$ ,  $\ell$ , k,  $\delta$ ,  $v$ ,  $\chi$ ,  $\sigma$ ) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return ( $\zeta$ ) ] ;

```

A return(*parametertreegenerate*(ξ , ρ , v ,*derivations*(v), ℓ ,k, δ , v , χ , σ)))

Note: In the above definition we must assume $\overline{\xi}$ is not \oplus .

```

parameterreegenerate (  $\xi, \rho, v, \pi, \ell, k, \delta, v, \chi, \sigma$  ) :=
  prog ( (  $\epsilon, R, y, \eta, \zeta, \psi$  ) ;
     $\epsilon := bestlist(\pi, k)$  ;
    [  $\epsilon = \emptyset \rightarrow return( overallcheck(\rho, \xi, \ell, \sigma, k))$  ] ;
     $R := rule(\epsilon)$  ;
     $y := variable(\epsilon)$  ;
    [  $\sim ad(y) \in proj_1(\ell) \rightarrow go(A)$  ] ;
     $\eta := \left( \bigcap a(y) \bigcap ad( find( ad(y), \ell ) ) \right)$  ;
    [  $\sim result(\bar{\rho}, result(\bar{R}, \eta)) = \bar{\xi} \rightarrow go(A)$  ] ;
     $\zeta := etacheck(\rho, R, \eta, \xi, \ell, k, \epsilon, y, \delta, v, \chi, \sigma)$  ;
    [ H-tagged(  $a(\zeta) \rightarrow return(\zeta)$  ) ] ;
A  $\psi := operate(punish(v, jumpback(v, \epsilon)))$  ;
  return ( parameterreegenerate (  $\psi(\xi), \psi(\rho), \psi(v), \psi(\pi) - \{\psi(\epsilon)\}, \psi(\ell), \uparrow k,$ 
     $\psi(\delta), \psi(v), \psi(\chi), \psi(\sigma))$  ) )

```

```

parametercheckgen (  $\rho, \omega, \xi, \ell, k, \delta, v, \chi, \sigma$  ) :=
  prog ( (  $\gamma, \phi$  ) ;
     $\gamma := constructderivation(\rho, \omega, requiredantecedents(\bar{\rho}, \omega, k), \xi)$  ;
     $\phi := operate(\gamma)$  ;
    return ( completederiv (  $\lambda, \phi(\delta),$ 
       $\phi( \ell \cup pair(\chi, antecedents(\gamma)), \{ \bigcap \delta \bigcap \gamma \}, \left( \bigcap ad(v) \bigcap ad(\omega) \right) \} ),$ 
       $\phi(\sigma), \uparrow k$  ) )

```

```

completederiv (  $\gamma, \delta, \ell, \sigma, k$  ) :=
  completederivv (  $\gamma, antecedents(\gamma), antecedents(\delta), \ell, \sigma, k$  )

```

```

completeredivv(  $\gamma$ ,  $\pi$ ,  $\tau$ ,  $\ell$ ,  $\sigma$ ,  $k$ ) =: prog(( $\zeta$ ,  $\phi$  ) ;
  [  $\pi = \emptyset \rightarrow$  return(H-tag(down( $\gamma$ ))) ] ;
   $\zeta :=$  prove( a( $\pi$ ), a( $\tau$ ),  $\ell$ ,  $\sigma$ ,  $k$ ) ;
  [  $\sim$  H-tagged( a( $\zeta$ ))  $\rightarrow$  return(down( $\gamma$ )) ] ;
   $\phi :=$  operate( a( $\zeta$ )) ;
  return(completeredivv( $\phi$ ( $\gamma$ ),  $\phi$ ( d( $\pi$ )),  $\phi$ (d( $\tau$ )),  $d$ ( $\zeta$ ),  $\phi$ ( $\sigma$ ),  $k$ )) )

overallcheck(  $\rho$ ,  $\xi$ ,  $\ell$ ,  $\sigma$ ,  $k$ ) =: prog(( $R$ ,  $\theta$ ,  $\zeta$ ,  $\phi$  ) ;
   $R :=$  bestnetparameterrule( $\rho$  ,  $k$ ) ;
  [  $R = \emptyset \rightarrow$  return( $\xi * \ell$ ) ] ;
   $\theta :=$  operate( $R$ ) ;
   $\zeta :=$  Rcheck( $\theta$ ( $\rho$ ),  $R$ ,  $\theta$ ( $\xi$ ),  $\theta$ ( $\ell$ ),  $k$ ,  $\emptyset$ ,  $\emptyset$ ,  $\sigma$ ) ;
  [ H-tagged( a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] !
   $\phi :=$  operate( erasepunish( $\rho$ , jumpback( $\rho$ ,  $R$ ))) ;
  return(overallcheck( $\phi$ ( $\rho$ ),  $\phi$ ( $\xi$ ),  $\phi$ ( $\ell$ ),  $\phi$ ( $\sigma$ ) ,  $\uparrow k$ )) )

Rcheck(  $\rho$ ,  $R$ ,  $\xi$ ,  $\ell$ ,  $k$ ,  $\delta$ ,  $v$ ,  $\chi$ ,  $\sigma$ ) =: prog(( $\eta$ ,  $\zeta$ ,  $\phi$  ) ;
   $\eta :=$   $\left( \left( \text{variablein}(\bar{R}) \right) \bowtie \text{bestnetparametervalue}(\text{proj}_2(\ell) \cup \{\sigma\}, k) \right)$  ;
  [ ad( $\eta$ ) =  $\emptyset \rightarrow$  return( $\xi * \ell$ ) ] ;
   $\zeta :=$  etacheck( $\rho$ ,  $R$ ,  $\eta$ ,  $\xi$ ,  $\ell$ ,  $k$ ,  $\emptyset$ ,  $\emptyset$ ,  $\delta$ ,  $v$ ,  $\chi$ ,  $\sigma$ ) ;
  [ H-tagged(a( $\zeta$ ))  $\rightarrow$  return( $\zeta$ ) ] ;
   $\phi :=$  operate( erasepunish( $\xi$ , jumpback( $\xi$ ,  $\eta$ ))) ;
  return( Rcheck( $\phi$ ( $\rho$ ),  $\phi$ ( $R$ ),  $\phi$ ( $\xi$ ),  $\phi$ ( $\ell$ ),  $\downarrow k$ ,  $\phi$ ( $\delta$ ),  $\phi$ ( $v$ ),  $\phi$ ( $\chi$ ),  $\phi$ ( $\sigma$ )) )

```

```

etacheck( $\rho, R, \eta, \xi, \ell, k, \varepsilon, \gamma, \delta, v, \chi, \sigma$ ) = prog( $m, \gamma, \phi, \zeta$ ) ;
  m :=  $\ell$ 
   $\gamma := \text{constructparameterderivation}(R, \eta, \langle \langle ad(\eta) \rangle \rangle, \text{result}(\bar{R}, \eta, k))$  ;
  [  $\varepsilon \neq \emptyset \rightarrow m := \ell \cup \{ \langle \langle \varepsilon \rangle \rangle, \gamma \}$  ] ;
  [  $\varepsilon \neq \emptyset \wedge \gamma \neq \emptyset \rightarrow m := \ell \cup \{ \langle \langle \varepsilon \rangle \rangle, \langle \langle ad(\gamma) \rangle \rangle, \langle \langle ad(\eta) \rangle \rangle \}$  ] ;
   $\phi := \text{operate}(\gamma)$ 
   $\zeta := \text{parametercheckgen}(\phi(\rho), \langle \langle a(v) \rangle \rangle, \langle \langle \text{down}(\gamma) \rangle \rangle, \phi(\xi), \phi(m),$ 
                                      $k, \phi(\delta), \phi(v), \phi(\chi), \phi(\sigma) )$  ;
  [H-tagged( $a(\zeta)$ )  $\rightarrow \text{return}(\zeta)$  ] ;
  return( $\xi * \ell$ )
  )

```

REFERENCES

- Amarel, Saul, "On the Automatic Formation of a Computer Program which Represents a Theory" in Self-Organizing Systems, edited by Yovits, Marshal C., et al., Washington, D.C.: Spartan Books, 1962.
- Church, Alonzo, Introduction to Mathematical Logic, Princeton, New Jersey: Princeton University Press, 1956.
- Friedberg, R. M., "A Learning Machine: Part 1," I.B.M. Journal of Research and Development 2, 1, 2-13 (1958).
- Holland, John H., "Adaptive Plans Optimal for Payoff-Only Environments". To appear in Proceedings of the Second Hawaii International Conference on System Science (1969).
- Holland, John H., "A Logical Theory of Adaptive Systems Informally Described," University of Michigan Engineering Summer Conferences (1961), pp. 1-51.
- McCarthy, John, Paul W. Abrams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, LISP 1.5 Programmers Manual, Cambridge, Massachusetts: Massachusetts Institute of Technology, Computation Center and Research Laboratory of Electronics, 1962.
- Mendelsohn, Elliott, Introduction to Mathematical Logic, Princeton, New Jersey: Van Nostrand, 1964.
- Newell, A., J. C. Shaw, and H. A. Simon, "Report on a General Problem Solving Program," University of Michigan Engineering Summer Conferences (1961), pp. 1-27.
- Polya, Gyorgy, How to Solve It, Princeton, New Jersey: Princeton University Press, 1945.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) LOGIC OF COMPUTERS GROUP 611 Church Street The University of Michigan, Ann Arbor, Mich.		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE A Self-Describing Axiomatic System as a Suggested Basis for a Class of Adaptive Theorem Proving Machines			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Technical Report			
5. AUTHOR(S) (Last name, first name, initial) Westerdale, Thomas H.			
6. REPORT DATE March, 1969	7a. TOTAL NO. OF PAGES 266	7b. NO. OF REFS 9	
8a. CONTRACT OR GRANT NO. DA-31-124-ARO-D-483	9a. ORIGINATOR'S REPORT NUMBER(S) 08226-7-T		
b. PROJECT NO.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)		
c.			
d.			
10. AVAILABILITY/LIMITATION NOTICES Distribution of This Document is Unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY U. S. Army Research Office (Durham) Durham, North Carolina	
13. ABSTRACT An explicitly self-describing axiomatic system is presented whose set of rules of inference continually increases in size as new theorems are proved. A proof of consistency relative to formal arithmetic is outlined. Modified LISP programs are the function constants of the system. A class of possible adaptive theorem proving machines is outlined. Such machines construct proofs by successively refining proof "outlines" which employ heuristics. New heuristics are generated by the same mechanism used to generate rules of inference and theorems. In the notation of the axiomatic system, a heuristic or a rule of inference is itself a well formed formula.			

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

INSTRUCTIONS

1. **ORIGINATING ACTIVITY:** Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization (*corporate author*) issuing the report.
- 2a. **REPORT SECURITY CLASSIFICATION:** Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.
- 2b. **GROUP:** Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.
3. **REPORT TITLE:** Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.
4. **DESCRIPTIVE NOTES:** If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.
5. **AUTHOR(S):** Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.
6. **REPORT DATE:** Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.
- 7a. **TOTAL NUMBER OF PAGES:** The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.
- 7b. **NUMBER OF REFERENCES:** Enter the total number of references cited in the report.
- 8a. **CONTRACT OR GRANT NUMBER:** If appropriate, enter the applicable number of the contract or grant under which the report was written.
- 8b, 8c, & 8d. **PROJECT NUMBER:** Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.
- 9a. **ORIGINATOR'S REPORT NUMBER(S):** Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.
- 9b. **OTHER REPORT NUMBER(S):** If the report has been assigned any other report numbers (*either by the originator or by the sponsor*), also enter this number(s).
10. **AVAILABILITY/LIMITATION NOTICES:** Enter any limitations on further dissemination of the report, other than those

imposed by security classification, using standard statements such as:

- (1) "Qualified requesters may obtain copies of this report from DDC."
- (2) "Foreign announcement and dissemination of this report by DDC is not authorized."
- (3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."
- (4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."
- (5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. **SUPPLEMENTARY NOTES:** Use for additional explanatory notes.
12. **SPONSORING MILITARY ACTIVITY:** Enter the name of the departmental project office or laboratory sponsoring (*paying for*) the research and development. Include address.
13. **ABSTRACT:** Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as (TS), (S), (C), or (U).

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. **KEY WORDS:** Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

UNIVERSITY OF MICHIGAN



3 9015 03627 7419