Stanford Department of Computer Science
Report No. STAN-CS-80-819

August 1980

COMPUTATIONAL USES OF THE MANIPULATION OF FORMAL PROOFS

by

Christopher Alan Goad

Research sponsored by

Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		*READ INSTRUCTIONS BEFORE COMPLETING FORM*
1. REPORT NUMBER STAN-CS-80-819	2. GOVT ACCESSION NO. AD-A091 180	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Computational Uses of the Manipulation of Formal Proofs.		5. TYPE OF REPORT & PERIOD COVERED technical, August 1980
		(14) 6. PERFORMING ORG. REPORT NUMBER STAN-CS-80-819
7. AUTHOR(s) (10) Christopher Alan Coad		(15) 8. CONTRACT OR GRANT NUMBER(s) MDA 903-80-C-0102, ✓ ARPA Order-2494
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (12) 133
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlignton, Virginia 22209		(11) 12. REPORT DATE August 1980 / 13. NO. OF PAGES 130
		15. SECURITY CLASS. (of this report) Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this report) (9) Doctoral thesis, Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) (see other side)		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

094120 LB

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Mechanical procedures for the manipulation of formal proofs have played a central role in proof theory for more than fifty years. However, such procedures have not been widely applied to computational problems. One reason for this is that work in computer science to do with formal proof systems has emphasized th- use of formal proofs as evidence – as tools for automatically establishing the truth of propositions. As a consequence of this emphasis, the problem for mchanizing the construction of proofs has received much attention, whereas the manipulation of proofs – that is, the conersion of one form of evidence into another – has not.

However, formal proofs can serve purposes other than the presentation of evidence. In particular, a formal proof of a proposition having the form, "for each *x* there is a *y* such that the relation *R* holds between *x* and *y*" provides, under the right conditions, a method for computing values of *y* from values of *x*. That is, such a proof describes an algorighm *A* where *A* satisfies the specification *R* in the sense that for each *x*, *R(x,A(x))* holds. Thus formal proof systems can serve as programming languages – languages for the formal description of algorithms. A proof which describes an algorithm may be "executed" by use of any of a variety of procedures developed in proof theory.

A proof differs from more conventional descriptions of the same algorithm in that it formalizes additional information about the algorithm beyond that formalized in the conventional description. This information expands the class of transformations on the algorithm which are amenabel to automation. For example, there is a class of "pruning" transformations which improve the computational efficiency of a natural deduction proof regarded as a program by removing unneeded case analyses. These transforations make essential use of dependency information which finds formal expression in a proof, but not in a conventional program. Pruning is particularly useful for removing redundancies which arise when a general purpose algorithm is adapted to a special situation by symbolic execution.

This thesis concerns (1) computational uses of the additional information contained in proofs, and (2) efficient methods for the representation and transformation of proofs. An extended lambda-calculus is presented which allows compact expression of the computationally significant part of the information contained in proofs. Terms of the calculus preserve dependency data, but can be efficiently executed by an interpreter of the kind used for lambda-calculus based languages such as LISP. The calculus has been implemented on the Stanford Artificial Intelligence Laboratory PDP-10 computer. Results of experiments on the use of pruning transformations in the specialization of a bin-packing algorithm are reported.

DD FORM 1 JAN 73 1473 (BACK)
EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Stanford Department of Computer Science
Report No. STAN-CS-80-819

August 1980

COMPUTATIONAL USES OF THE MANIPULATION OF FORMAL PROOFS

by

Christopher Alan Goad

ABSTRACT

Mechanical procedures for the manipulation of formal proofs have played a central role in proof theory for more than fifty years. However, such procedures have not been widely applied to computational problems. One reason for this is that work in computer science to do with formal proof systems has emphasized the use of formal proofs as evidence – as tools for automatically establishing the truth of propositions. As a consequence of this emphasis, the problem for mchanizing the construction of proofs has received much attention, whereas the manipulation of proofs – that is, the conersion of one form of evidence into another – has not.

However, formal proofs can serve purposes other than the presentation of evidence. In particular, a formal proof of a proposition having the form, "for each x there is a y such that the relation R holds between x and y" provides, under the right conditions, a method for computing values of y from values of x. That is, such a proof describes an algorighm A where A satisfies the specification R in the sense that for each x, $R(x,A(x))$ holds. Thus formal proof systems can serve as programming languages – languages for the formal description of algorithms. A proof which describes an algorithm may be "executed" by use of any of a variety of procedures developed in proof theory.

A proof differs from more conventional descriptions of the same algorithm in that it formalizes additional information about the algorithm beyond that formalized in the conventional description. This information expands the class of transformations on the algorithm which are amenabel to automation. For example, there is a class of "pruning" transformations which improve the computational efficiency of a natural deduction proof regarded as a program by removing unneeded case analyses. These transforations make essential use of dependency information which finds formal expression in a proof, but not in a conventional program. Pruning is particularly useful for removing redundancies which arise when a general purpose algorithm is adapted to a special situation by symbolic execution.

This thesis concerns (1) computational uses of the additional information contained in proofs, and (2) efficient methods for the representation and transformation of proofs. An extended lambda-calculus is presented which allows compact expression of the computationally significant part of the information contained in proofs. Terms of the calculus preserve dependency data, but can be efficiently executed by an interpreter of the kind used for lambda-calculus based languages such as LISP. The calculus has been implemented on the Stanford Artificial Intelligence Laboratory PDP-10 computer. Results of experiments on the use of pruning transformations in the specialization of a bin-packing algorithm are reported.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-80-C-0102. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

to my parents

Walter B. Goad

and

Maxine S. Goad

Acknowledgements

I wish to thank my adviser, Professor Kreisel, for the help and guidance which he has given me. Kreisel introduced me to proof theory, and later persuaded me to experiment with the manipulation of proofs by computer. Throughout the preparation of this dissertation, Kreisel provided ideas, criticisms, and encouragement which were of great value to me. His help in the writing phase was extensive; the dissertation as it now stands incorporates a large number of Kreisel's detailed suggestions concerning organization and style. More generally, I wish to thank Kreisel for demonstrating by his example that precise thought is possible about matters which have not yet received a technical formulation, and that there is no need to cast away one's ordinary judgement when in the presence of a technical definition.

I would not have studied proof theory, and certainly would not have undertaken the work described in this thesis, had it not been for the influence and help of Professor John McCarthy, and of Dr. Richard Weyhrauch. I originally developed an interest in applications of logic to computing while working on the FOL project at the Stanford Artificial Intelligence Laboratory under their supervision. Professor McCarthy made it possible for me to pursue this interest by continuing to support me as member of the Stanford Artificial Intelligence Project while the work described in this thesis was in progress. I am very grateful for that support and also for the encouragement which Professor McCarthy has given me throughout my stay at Stanford. I wish to thank Dr. Weyhrauch for the many useful and enjoyable discussions which I have had with him over the past five years. I am grateful to both Professor McCarthy and Dr. Weyhrauch for serving on my reading committee.

The Stanford Artificial Intelligence Laboratory provided an ideal environment for doing research - both in terms of computing resources and intellectual stimulation. In addition, it was a very nice place to work (this was particularly true when it had a separate physical existence in the D.C. Power building in the hills above Stanford). I wish to thank everyone associated with the laboratory.

I wish to thank my wife Elspeth - not for any influence she has had on the content of the thesis, but rather for her effect on the happiness of its author.

Finally, I wish to thank IBM corporation for the graduate fellowship which supported my studies during the 1977-78 academic year.

This text of this thesis was prepared on an Alto computer donated to the Stanford Computer Science Department by Xerox corporation.

Abstract

Mechanical procedures for the manipulation of formal proofs have played a central role in proof theory for more than fifty years. However, such procedures have not been widely applied to computational problems. One reason for this is that work in computer science to do with formal proof systems has emphasized the use of formal proofs as evidence - as tools for automatically establishing the truth of propositions. As a consequence of this emphasis, the problem of mechanizing the construction of proofs has received much attention, whereas the manipulation of proofs - that is, the conversion of one form of evidence into another - has not.

However, formal proofs can serve purposes other than the presentation of evidence. In particular, a formal proof of a proposition having the form, "for each x there is a y such that the relation R holds between x and y" provides, under the right conditions, a method for computing values of y from values of x. That is, such a proof describes an algorithm A where A satisfies the specification R in the sense that for each x, R(x,A(x)) holds. Thus formal proof systems can serve as programming languages - languages for the formal description of algorithms. A proof which describes an algorithm may be "executed" by use of any of a variety of procedures developed in proof theory.

A proof differs from more conventional descriptions of the same algorithm in that it formalizes additional information about the algorithm beyond that formalized in the conventional description. This information expands the class of transformations on the algorithm which are amenable to automation. For example, there is a class of "pruning" transformations which improve the computational efficiency of a natural deduction proof regarded as a program by removing unneeded case analyses. These transformations make essential use of dependency information which finds formal expression in a proof, but not in a conventional program. Pruning is particularly useful for removing redundancies which arise when a general purpose algorithm is adapted to a special situation by symbolic execution.

This thesis concerns (1) computational uses of the additional information contained in proofs, and (2) efficient methods for the representation and transformation of proofs. An extended lambda-calculus is presented which allows compact expression of the computationally significant part of the information contained in proofs. Terms of the calculus preserve dependency data, but can be efficiently executed by an interpreter of the kind used for lambda-calculus based languages such as LISP. The calculus has been implemented on the Stanford Artificial Intelligence Laboratory PDP-10 computer. Results of experiments on the use of pruning transformations in the specialization of a bin-packing algorithm are reported.

CONTENTS

Chapter 1

Introduction

The most obvious purpose of a proof is to convince - to provide compelling evidence for the truth of a proposition. A *formal* proof provides evidence of a kind that can be mechanically recognized, and it is in the capacity of evidence that formal proofs have most often been used in computation, as for example in automatic theorem proving and in automatic program verification of the usual kind.

As a consequence of the emphasis on the use of proofs as evidence, only two of the various operations which people commonly perform on informal proofs have played a significant role in computations involving formal proofs. These operations are the construction of proofs, and the checking or recognition of proofs. Operations which involve the actual manipulation of existing proofs, as opposed to the manipulation of formulas, are not much used.

However, mechanical procedures for proof manipulation have played a central role in the subfield of mathematical logic known as proof theory for more than fifty years. This thesis concerns applications of proof theoretic methods to computational problems. In particular, our subject matter is the use of formal proofs for the description of algorithms, and the transformations on algorithms which are made possible by this mode of description. Thus the work differs from most work in computer science to do with formal proofs both in the use to which proofs are put, and in the emphasis placed on the manipulation - in contrast to the construction - of proofs.

The manner in which proofs may be used to express algorithms is as follows. Suppose that one has a proof that an object with given properties exists. Then the proof can sometimes be used to discover the identity of a particular object with those properties. If restrictions are made on the forms of inference used, then it is possible to guarantee that the proof will (in one sense or another) provide this additional information. For example, a *constructive* proof of $\exists x\varphi(x)$ always "provides" a value v with $\varphi(v)$ in the sense of indicating a method for computing v; the computation may or may not be feasible in practice. However, the restriction to constructivity is too strong. For one thing, a proof of $\exists x\varphi(x)$ may exhibit a value v which satisfies φ, but show that $\varphi(v)$ holds by non-constructive methods. Also, if one restricts the complexity of φ (for example, if φ is a quantifier free formula of first order arithmetic), then any classical proof of $\exists x\varphi(x)$ will provide a realization in the same sense and by the same formal methods as a constructive proof. (By a "realization" of an existential statement $\exists x\varphi(x)$ is meant simply a value which satisfies the predicate φ.)

If an existence proof is given in a formal way - in a way which makes it suitable for mechanical manipulation - then one might hope to mechanize the passage from the proof to the value realizing the existential statement. Work in proof theory has shown that the extraction of realizations from proofs can in fact be mechanized for a variety of formal systems and in a variety of ways. For example, Prawitz's normalization procedure may be used to transform a natural deduction proof of an existential formula into a direct proof of the same formula which will - under rather general conditions - explicitly mention a realization.

Now, if one has a proof of a formula of the form $\forall x\exists y\varphi(x,y)$, the methods from proof theory mentioned just above can evidently be used to compute a function f with $\forall x\varphi(x,f(x))$. To do this, simply apply the general result $\forall x\exists y\varphi(x,y)$ to the input value, and then use normalization (or whatever method one has in hand) to extract a realization. Thus a proof of a formula $\forall x\exists y\varphi(x,y)$ serves the role of a program which computes a function satisfying the "specification" φ.

Given that proofs can be used as programs, what is the interest of this fact for computer science and for practical computing? One answer is as follows.

Existing programming languages are for the most part designed with economy of expression in mind; a program in such a language formalizes exactly the information needed for carrying out the task at hand. A proof, on the other hand, formalizes a great deal of information which is not essential for the simple execution of a computation - such as a description of the task being performed, a verification of the method, and an account of the dependencies between facts involved in the computation. The additional information contained in proofs is useful in the transformation of computing methods - for example in adapting methods to new situations. This should not be surprising, since one expects that the data relevant to the transformation of algorithms will be different and more extensive than the data needed for simple execution.

We shall be concerned with a particular set of transformations on algorithms - called the "pruning transformations". These transformations remove redundant chunks of computation by making use of a kind of dependency information which does not appear in ordinary programs. For the most part, the redundancies removed by pruning are not to be found in proofs generated by people. Thus the pruning transformations will not be of much use when applied to algorithms as originally presented. However, proofs which result from automatic processes tend to include such redundancies.

For example, suppose that one has an algorithm A(x) which is to be used in a situation where it is known in advance that all inputs will have a special form given by the term $t(y_1, \ldots y_n)$. Then A may be automatically adapted to perform efficiently in this special situation by symbolically executing the code for A on the term t, and then applying optimizing transformations to the result. (Ershov[1977] and Sandewall[Beckeman, Haraldsson, Oskarsson, and Sandewall, 1976], among others, have studied this method of specialization as it applies to ordinary programs.) If A is expressed by a proof Π, then the result of symbolically executing Π on the term t will often contain redundancies of the kind removed by pruning even if Π as originally given contained no such redundancies. Thus, the effectiveness of automatic specialization can be increased by adding pruning to the arsenal of optimizations used in the course of specialization.

As they stand, the standard methods of proof theory are not adequate for carrying out the specialization of algorithms in a feasibly efficient way. However, we have devised methods for the execution and pruning of proofs which overcome this problem. The methods have been implemented in a proof manipulation system running on the Stanford Artificial Intelligence Laboratory PDP-10 computer. As a preliminary empirical investigation of the usefulness of pruning in the specialization of algorithms, expermiments on the specialization of a bin-packing algorithm have been carried out.

The following topics are treated in this thesis, listed in order of decreasing generality.

(1) the use of proofs for the formalization of algorithms,

(2) optimizing transformations on proofs, in particular, the pruning transformations,

(3) efficient implementation of operations on proofs,

(4) the use of pruning in the specialization of algorithms, and

(5) the specialization of a bin-packing algorithm.

The general objective of the work is the development of an improved technology for the manipulation of algorithms. The use of enriched formal descriptions of algorithms - specifically, formal proofs - is a means to this end.

The contents of the thesis are as follows. Chapter 2 serves to introduce some material from proof theory which will be needed in the course of the thesis. In particular, we define the notion of a natural deduction proof system, and explain Prawitz's normalization procedure. Also, we present a very simple example of the use of pruning in specializing algorithms. The example is intended to illustrate the central features of the pruning transformations in a setting of minimal technical complexity. Chapter 3 describes the methods which we have devised for the efficient execution and pruning of proofs. In chapter 4, results of the bin-packing experiments are reported. Chapter 5 sketches additional uses which might be made of the proof technology described in chapter 3. There are two appendices, each of which is intended primarily for readers with an interest in traditional proof theory. The first concerns the relationship between our methods and the functional and realizability interpretations of Kleene, Gödel, and Kreisel. The second appendix presents an example which demonstrates that the features of proof systems which are of interest for traditional proof theory are different from those which are most directly relevant to the computational use of proofs.

The remainder of this chapter is devoted to a collection of general remarks about the work, and to previews of matters which are discussed in detail later on.

° *Manipulation vs. construction*

It should be emphasized again that the work described in this paper concerns the automatic *manipulation* of existing proofs, and not the automatic construction of new proofs. The bin packing proof used in the experiments was devised "by hand", and was entered by hand into the proof checking component of the proof manipulation system. If one is able to automate, fully or partially, the construction of proofs which describe computational methods, then so much the better. But such matters lie outside the scope of this thesis.

° *Differences between proofs used to describe computation and proofs used as evidence*

It is necessary to keep computational considerations explicitly in mind when constructing proofs which are intended as descriptions of computation. The best proof of a formula $\forall x \exists y \varphi(x,y)$ according to such standard criteria as brevity, elegance or comprehensibility, will often embody a very bad algorithm. Conversely, a proof of $\forall x \exists y \varphi(x,y)$ which formalizes a good algorithm will generally constitute a rather unnatural way of establishing the simple truth of the formula. For the purposes of this thesis, proofs are to be regarded as a means of formulating algorithmic ideas. In writing a proof to be used for solving a computational problem, one follows the same procedure as is used in writing an ordinary program. Namely, one first devises a reasonable algorithm, and afterwards formalizes that algorithm (as a proof). If a proof is given in complete detail, then it includes a justification for the correctness of the algorithm which it formalizes. As an immediate consequence, formalization of algorithms by proofs provides a means for the mechanical verification of algorithms.

However, if one wishes only to implement an algorithm, and not to verify it, then the proof describing the algorithm need not be fully formalized. In particular, proofs of so-called "Harrop formulas" can be left out. The Harrop formulas include for example all formulas which lack occurences of the positive logical symbols ∨ and ∃. Any proof of a Harrop formula may be omitted without destroying the computational usefulness of a proof in which that axiom appears.

Such "non-computational" formulas do not even need to be true. A proof which uses incorrect Harrop formulas as axioms can be executed and pruned in the same manner as a proof which is valid throughout. However, the func'ion computed by the incorrect proof may not satisfy the specification embodied in its end-formula.

A formal proof which is constructed for the purpose of describing an algorithm and which makes free use of Harrop formulas as axioms will in general contain only a part of the information needed to establish the truth of its end-formula. Thus the formal proofs which will concern us here are not proofs in the ordinary sense at all; they do not supply - and are not intended to supply - the evidence necessary to verify a proposition. We are bending the machinery of formal proofs to a different end than that for which it was originally intended, and so can discard the part of that machinery which is irrelevant to our new purposes.

° *The role of constructive methods*

We restrict our attention in this thesis to proofs which are built up using constructively valid inferences. The particular formal proof system used is the natural deduction formulation of first order logic as originally developed by Gentzen[1969] and later studied by Prawitz[1965]. To arrive at the constructive (or "intuitionistic") variant of natural deduction from the standard or classical natural deduction system for first order logic , one simply removes one of the inference rules, namely the rule which expresses the principle of the excluded middle.

Note for the reader who is unfamiliar with intuitionistic logic: The approach to the foundations of mathematics which is known as "intuitionism" or "constructivism" was originated by Brouwer. According to this approach, the subject matter of mathematics is not an external world of mathematical objects, but rather the world of mental constructions carried out by mathematicians. This point of view leads to a reinterpretation of the meanings of the logical symbols, and to restrictions on the modes of inference which can be employed. Heyting and later Gentzen developed formal systems for representing contructive reasoning. It is not our intention here to give an exposition of intuitionism as a philosophical standpoint; the interested reader is referred to van Dalen [1973].

We have chosen to use the the constructive instead of the standard system not because of any distrust of classical reasoning, nor because non-constructive proofs cannot be used to describe algorithms. Indeed, the proofs which we use to describe algorithms will in any case make use of complicated axioms (as explained in the last section), and there is no reason whatever to require that these axioms be constructively valid. Thus the formulas which appear in our proofs will not in general be constructively valid; it is only the inference rules used for manipulating those formulas which must be constructive. But further, even proofs which make essential use of non-constructive inferences in connection with non-Harrop formulas can be executed by methods similar to those used for constructive proofs. In particular, many of the methods of proof theory, including Prawitz's normalization method, apply to classical proofs as well as to constructive proofs, and under certain conditions are guaranteed to provide the same kind of information. For example, normalization may be used to execute any (classical) proof of a formula $\forall x \exists y \varphi(x,y)$ of arithmetic whose matrix φ is quantifier free; a value for y will always be supplied by normalization when any input value for x is given. Thus the distinction between a proof which describes an algorithm and a proof which does not is quite different from the distinction between a constructive and a non-constructive proof.

However, the process of fleshing out an algorithm into a proof from (possibly complex) Harrop axioms appears to lead naturally to a proof in which only constructive inferences are used. This at least is our experience so far. So for the moment, there is no need to look at classical systems, and by the restriction to constructive systems we are able to avoid a certain amount of technical complication.

° *The p-calculus*

Traditional proof theory provides two kinds of methods for the execution of proofs. First, there are the normalization and cut-elimination methods which carry out the computation indicated by a proof by transformation of the proof itself. Second, there are the functional and realizability interpretations which extract "code" of one kind or another from proofs; it is then the code which is executed, and not the proof itself.

Each of these two approaches is inadequate for the purposes which we have in mind here. The normalization methods are unsatisfactory because they are too slow. On the other hand, the methods which involve extraction of code from proofs retain only the information which is needed for the computation immediately at hand; the additional data needed for the pruning transformations is lost. This would not be a problem if we only intended to apply pruning transformations to proofs as they are originally given. However, the use of proofs for the specialization of algorithms requires that the additional data be preserved by symbolic execution.

Our solution to these difficulties involves the use of an extended λ-calculus, which we shall refer to as the p-calculus. The p-calculus is designed to provide expression for just that information contained in natural deduction proofs which is needed for execution and for the pruning operations. P-calculus terms can be extracted from ordinary natural deduction proofs in a straight-forward manner, and executed efficiently by an interpreter of the kind used for λ-calculus based languages such as LISP and SCHEME. Chapter 3 describes the p-calculus in detail.

° *Related work in computer science*

The work described in this thesis is related in a general way to work in a variety of areas of computer science. In particular, there are clear connections to code optimization, program synthesis and transformation, and to dependency directed reasoning in the sense of [London 1978] and [Stallman & Sussman 1977]. The relation between the current work and the topics just mentioned is discussed in chapter 5. In what follows, we give a brief catalog of work within computer science which is directly concerned with the extraction of information from proofs.

Green [1969] considered the problem of extracting information from resolution proofs. Bishop[1970], Constable[1971], and Martin-Löf[1979] - among others - have suggested using constructive proof systems as programming languages. Goto[1979] has implemented Gödel's Dialectica interpretation for intuitionistic first-order arithmetic. Takasu [1978] discusses computational uses of proofs in the same system by use of Gentzen's [1969] cut-elimination procedure. Miglioli and Ornaghi [1980] describe a method for executing sequent calculus proofs which differs from cut-elimination. In [Manna and Waldinger, 1979], a method for automatic synthesis of programs is described which involves the simultaneous construction of a natural deduction proof of the goal formula and of a program which realizes that formula (in a suitable sense). Bates [1979] develops a constructive "refinement logic", and shows how programs can be extracted from proofs of this logic. A *Prolog* program [Kowalski 1974] is a collection of axioms in Horn clause form. An execution of a Prolog program consists of a search for a proof in a restricted resolution system. The output is a term extracted from the proof. In practice, the output term is constructed during the search for the proof. (See chapter 5 for further comments concerning the work of Bates and of Kowalski.)

It should be emphasized that the aims of the work described just above differ fundamentally from the aims of the work presented in this thesis. In the former, formal proofs serve as vessels from which computational contents of a standard kind are extracted. In contrast, our concern is to exploit the *differences* between proofs and conventional descriptions of computation. Specifically, we will show how new operations on algorithms can be mechanized by making use of the additional information to be found in proofs.

Chapter 2

Normalization and Pruning of Natural Deduction Proofs

In this chapter, we describe the natural deduction formalism (section 2.1), and the normalization and pruning operations (sections 2.2, 2.7). A very simple example of the use of pruning in specializing algorithms is given in section 2.8. Our presentation of natural deduction and of normalization follows standard lines (eg Prawitz[1965]), except in the treatment of "lemmas" (section 2.5). Certain formal details concerning normalization are left out, and all results are stated without proof. Also, no treatment of principles of induction is given until Chapter 3, where normalization and pruning are described in formal detail as they apply to a computationally efficient representation of natural deduction proofs.

2.1 Natural deduction

Systems of natural deduction were originally developed by Gentzen[1969]. The notation used here is that of Prawitz[1965]. The reader is referred to Prawitz[1965] for a more discursive presentation of natural deduction and of a normalization procedure for natural deduction proofs.

In what follows, we describe the natural deduction formalism for intuitionistic first order logic. The formalism is defined with a first order language L as a parameter; the class of formulas which may appear in a proof is given by L. It should be noted that natural deduction differs from other proof systems for intuitionistic first order logic in the kind of structure which it provides for representing proofs, and not, for example, in the set of theorems which it proves. It is possible to translate back and forth between proofs of natural deduction and proofs of, say, the sequent calculus in a mechanical way. The advantages of natural deduction are the advantages of a good data structure - a data structure which represents human reasoning in a comparatively direct way, and to which the various operations in which we are interested can be easily applied.

The notion of a first order language is defined in the standard manner, as follows. We start with (1) an (infinite) list of variable symbols, $v_1, v_2, \ldots$, (2) a list of constant symbols $c_1, c_2, \ldots$, (3) a list of relation symbols $R_1, R_2, \ldots$, and (4) a list of function symbols $f_1, f_2 \ldots$. The arities of the relation symbols and function symbols are to be specified as part of the definition of L. Terms of L are built up from variable and constant symbols by means of function application in the standard manner. The set of formulas of L is defined by the following inductive clauses. (1) The propositional constant FALSE is a formula.

(2) If $t_1, \ldots t_n$ are terms, and R is a relation symbol of arity n then $R(t_1, \ldots t_n)$ is a formula. (3) If P,Q are formulas, and x is a variable, then (a) $P \wedge Q$, (b) $P \vee Q$, (c) $P \supset Q$, (d) $\exists x P$ are formulas. It is convenient for our purposes to allow universal quantification to apply to a vector of variables; thus we have (e) $\forall x_1, \ldots x_n P$ is a formula for any formula P and vector of distinct variables $x_1, \ldots x_n$. ($\forall x_1, \ldots x_n P$ is *not* an abbreviation for $\forall x_1 \forall x_2 \ldots \forall x_n P$. We shall sometimes use underlined characters to refer to vectors - for example $\underline{x}$ wi'' refer to a vector of variables, and $\underline{t}$ to a vector of terms. We regard negation as a defined notion; specifically, $\neg P$ is to be read as an abbreviation for the formula $P \supset$ FALSE. The notion of a free occurence of a variable in a formula is defined in the standard manner.

A natural deduction proof takes the form of a tree whose nodes are labeled by formulas, by the names of inference rules, and by other information. This tree represents the history of a logical argument - in particular it records a series of applications of inference rules which lead from the hypotheses of the argument (represented by leaf nodes of the tree) to its conclusion (represented by the root).

The leaves of a natural deduction proof tree are of two kinds: axioms and assumptions. The truth of the conclusion of a natural deduction proof will in general depend on the truth of the formulas which appear as axiom leaves, but may not depend on the truth of all of the formulas which appear as assumption leaves. The reason for this is that the inference rules of natural deduction can have the effect of "discharging assumptions". For example, consider the implication introduction rule:

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \supset B}$$

This rule specifies that $A \supset B$ can be inferred from B. In addition, the rule indicates that the set of assumptions upon which $A \supset B$ depends is to be computed by removing the formula A from the set of assumptions on which B depends. (The appearance of A in parentheses is what specifies that the assumption A is to be discharged.) Informally, the rule states that if B can be proved using the assumption A, then $A \supset B$ can be concluded, and this conclusion does not depend on A being true. Thus the inference rules of natural deduction operate not just on end-formulas of the subproofs to which they are applied, but on additional information contained in those subproofs, namely, sets of assumptions.

In general, the formula attached to any node in a natural deduction proof tree depends on some (possibly empty) subcollection of the formulas attached to assumption leaves of the subtree rooted at that node. The members of this subcollection are referred to as the "open assumptions" of the node. The inference rules specify what conclusions may be drawn from premises of a given form, and in addition indicate how the open assumptions of the conclusion are computed from the open assumptions of the premises.

The set of open assumptions of each node in a proof tree is computed recursively as follows. First of all, the set of open assumptions of a leaf node is the empty set if the node is an axiom, and the singleton set containing the node itself if the node is an assumption. The set of open assumptions of any non-leaf node can be computed from open assumptions of its sons simply by appying the inference rule associated with that node.

Note that we use the phrase "open assumptions" to refer to a set of nodes on a proof tree, and not to the set of formulas attached to those nodes.

Each of the inference rules of natural deduction has the following form:

$$\frac{\begin{matrix}(A_1) & (A_2) & \ldots & (A_n)\\ P_1 & P_2 & \ldots & P_n\end{matrix}}{C}$$

In the above, some (or all) of the P_i may lack associated appearances of parenthesized formulas (A_i). The meaning of such a rule is that a conclusion of form C can be derived from premise formulas of forms $P_1 \ldots P_n$. The set of open assumptions of the conclusion is computed as follows. *Let S_i be the set of open assumptions of premise P_i.* For each i, remove from S_i all nodes whose attached formula is A_i, and call the result S_i'. (If there is no A_i associated with P_i, then let $S_i' = S_i$.) The set of open assumptions of the conclusion is just the union of the S_i'. The A_i are called the assumptions discharged by the rule.

Each of the inference rules of natural deduction is devoted to the treatment of a particular logical symbol or quantifier. Conversely, for each logical symbol and quantifier, there is a rule (or pair of rules) which has the effect of introducing that symbol, and another rule (or pair of rules) which has the effect of eliminating that symbol. The rules are designated by the symbol which they treat, and by their function, whether it be introduction or elimination. For example, the two rules which treat implication are referred to as the "$\supset$-introduction rule" and the "$\supset$-elimination rule" ("$\supset$I" and "$\supset$E" for short).

The inference rules of natural deduction are given below. We use the following notation for substitution: $A_{[x\leftarrow t]}$ or $A[x\leftarrow t]$ denotes the result of replacing all occurences of the variable x by the term t in the formula A. If $\underline{x}$ and $\underline{t}$ are vectors of variables of the same length, then $A[\underline{x}\leftarrow\underline{t}]$ denotes the result of substituting the terms $\underline{t}$ for the variables $\underline{x}$ *in parallel*. As usual, substitution may require that bound variables be renamed.

$\wedge$-introduction:

$$\frac{A \quad B}{A \wedge B}$$

$\wedge$-elimination:

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$\vee$-introduction:

$$\frac{A}{A \vee B} \qquad \frac{B}{A \vee B}$$

$\vee$-elimination:

$$\frac{A \vee B \quad \begin{matrix}(A)\\ C\end{matrix} \quad \begin{matrix}(B)\\ C\end{matrix}}{C}$$

$\supset$-introduction:

$$\frac{\begin{matrix}(A)\\ B\end{matrix}}{A \supset B}$$

$\supset$-elimination:

$$\frac{A \quad A \supset B}{B}$$

$\forall$-introduction:

$$\frac{A}{\forall \underline{x} A}$$

condition: none of the variables $\underline{x}$ may appear free in any assumption on which the premise A depends.

∀-elimination:

$$\frac{\forall \underline{x} A}{A[\underline{x} \leftarrow \underline{t}]}$$ where $\underline{t}$ is any vector of terms of L

∃-introduction:

$$\frac{A[x \leftarrow t]}{\exists x A}$$ where t is any term of L

∃-elimination:

$$\frac{\exists x A \quad \overset{(A)}{C}}{C}$$

conditions: the variable x may not appear free in A, nor in C, nor in any assumption on which the second premise C depends other than the assumption A.

The above rules are essentially Prawitz's rules for the intuitionistic predicate calculus. However, we have left out the FALSE-elimination rule:

FALSE-elimination:

$$\frac{\text{FALSE}}{A}$$

The effect of this rule can be obtained by the use of axioms of the form FALSE $\supset$ A for atomic formulas A. (Any formula can be derived from FALSE by means of such axioms and the use of introduction rules. For example, $A \vee B$ with A atomic may be derived from FALSE by using the axiom FALSE $\supset$ A, and then applying $\vee$-introduction.) As will be seen (section 2.3), we shall allow such "false-elimination" axioms to appear in proofs used for computation; in fact, the restriction that the consequent A be atomic may be weakened - A may be any "Harrop" formula (section 2.3).

The classical first order predicate calculus is arrived at by adding the following inference rule expressing the principle of the excluded middle (recall that $\neg A$ abbreviates $A \supset FALSE$).

$\neg$-elimination:

$$\dfrac{\begin{array}{c}(\neg A)\\ FALSE\end{array}}{A}$$

Notice that free variables which appear in axioms are in effect universally quantified; the same conclusions can be drawn from an axiom $A(x_1, \ldots x_n)$ in which the x_i appear free as from the axiom $\forall x_1 x_2 \ldots x_n A(x_1, \ldots x_n)$.

The $\forall$-introduction and $\exists$-elimination inferences bind variables in a proof in the same sense that the quantifiers $\forall$ and $\exists$ bind variables in a formula. Specifically, the variables $\underline{x}$ in the above presentation of the $\forall$-introduction rule are to be regarded as bound wherever they occur in the proof of the premise of the rule. Similarly, the variable x in the $\exists$-introduction rule is to be regarded as bound in the proof of the rule's second premise. In both formulas and proofs, a bound variable serves as a local name which is meaningful only inside the scope of the binding; such bound variables may be renamed at will without changing the meaning of a formula or proof (as long as conflicts with other variable names are avoided). A precise definition of the notion of a bound variable in a proof will be given in chapter 3.

By a "closed proof" we mean a proof in which no variables occur free, and in which the end-formula depends on no assumption. Formulas which are not closed may appear in a closed proof, as long as the free variables in those formulas are bound by one of the inference rules $\forall$-introduction and $\exists$-elimination.

The following is a simple example of a natural deduction proof. The proof makes use of no axioms. Assumption leaves of the proof tree appear in brackets. The reader can verify that each of the assumptions is discharged in the course of the proof. The result is an assumption free derivation of the predicate caluclus theorem, $\forall y(P(y) \vee Q(y)) \supset \forall x(Q(x) \vee P(x))$.

$$\supset\! I\;\dfrac{\forall I\;\dfrac{\vee E\;\dfrac{\forall E\;\dfrac{[\forall y(P(y)\vee Q(y))]}{P(x)\vee Q(x)} \qquad \vee I\;\dfrac{[P(x)]}{Q(x)\vee P(x)} \qquad \vee I\;\dfrac{[Q(x)]}{Q(x)\vee P(x)}}{Q(x)\vee P(x)}}{\forall x(Q(x)\vee P(x))}}{\forall y(P(y)\vee Q(y)) \supset \forall x(Q(x)\vee P(x))}$$

2.2 Normalization

In the course of this thesis we will have occasion to consider several procedures for the step-by-step reduction of objects to a "normal" form. These "normalization" procedures share certain general features. This section introduces the basic notions and terminology which apply to normalization in each of its various forms.

The two standard normalization procedures which are most directly relevant to our current purposes are the proof normalization procedure of Prawitz, and the normalization procedure for Church's[1941] λ-calculus. The methods described in chapter 3 make essential use of the close connection between these two procedures.

Let T be a class of terms (of whatever kind). A normalization procedure for T is (partly) given by a collection R of "small" transformations, called "reduction rules". The normalization of a term t consists of repeated application of the reduction rules until no further application of a rule is possible. The result of this process (if it terminates) is called a "normal form of t", and is designated by $|t|$.

More precisely given a term t and a reduction rule r, r may or may not be applicable to t. If r is applicable to t, it may be applicable in various ways (in the case of proofs and λ-terms, the reduction rule may be applicable at several places within the proof or term). The result of applying a reduction rule in a particular way to a term t is a modified term t'. A term to which no reduction rule is applicable is said to be in normal form. A pair $\langle T,R\rangle$ where T is a set of terms and R a set of redution rules on those terms will be referred to as a "reduction system".

We use the notation $t_1 \rightarrow t_2$ to signify that t_2 results from an application of one of the reduction rules to t_1. Any procedure for selecting a particular order (and "way") in which reductions are to be applied to a term is called a "normalization procedure". Thus a normalization procedure, when applied to any particular term t generates a (possibly infinite) sequence of terms t_0,t_1,t_2 . . . where t_{i+1} is arrived at from t_i by the application of one of the reduction rules. A theorem which states that a given normalization procedure always yields a finite sequence of terms t_1 ,t_2, . . . t_n, where t_n is in normal form, regardless of the initial term t_1, is referred to as a "normalization theorem". Other standard terminology concerning normalization is as follows.

° A system $\langle T,R\rangle$ has the "termination" property if every sequence of reductions t_1,t_2 . . . is finite.

° We use the notation $t \rightarrow^* t'$ to signify that t' results from t by some finite sequence $t \rightarrow t_1 \rightarrow t_2 \rightarrow$ . . . t' of applications of reduction rules. A system $\langle T,R\rangle$ has the "uniqueness property" if every sequence of reductions of a term to a normal form yields the same result. That is, $\langle T,R\rangle$ has the uniqueness property if, whenever $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$, and t_1 and t_2 are both normal, then $t_1 \approx t_2$.

° A system which has both the termination and the uniqueness properties is said to have the "strong normalization" property. Evidently, if a system has the strong normalization property, then the normal form $|t|$ of each term exists and is unique.

Each of the computation procedures to be considered in the course of this thesis takes the form of a normalization procedure of one kind or another. Of course, normalization procedures need not be implemented in a literal minded way. Normalization for λ-calculus based languages can be sped up by using environments instead of literal substitution for carrying out λ-conversions. The implemented p-calculus interpreter on which the experiments were carried out makes use of this idea.

2.3 Computing using proof normalization

This section concerns the manner in which proof normalization may be used for computing, and not the internal workings of the normalization procedure itself.

The usefulness of proof normalization for computational purposes derives from the special properties possessed by proofs which are in normal form. Roughly speaking, the reductions used in proof normalization have the effect of removing certain kinds of indirect arguments from a proof. A normal proof contains none of these indirect forms of argument, and computationally useful information can be read off a proof which is direct in this sense.

Evidently, some restriction must be made on the axioms which appear in a proof if it is to be of any computational use. The appropriate restriction for our purposes is that all axioms be "Harrop formulas". The Harrop formulas are those which do not contain the positive logical symbols $\vee$ and $\exists$ except in the hypotheses of implications. More formally, the class of Harrop formulas is defined by the following inductive clauses: (a) atomic formulas are Harrop formulas, (b) if A and B are Harrop formulas, then so are $A \wedge B$, $\forall \underline{x} A$, (c) if B is a Harrop formula, then so is $A \supset B$, regardless of the form of A. A proof which contains only Harrop formulas as axioms will be referred to as a Harrop proof.

(The notion of a Harrop formula was introduced by Harrop[1960]. Harrop showed that if A and $\exists x B(x)$ are closed formulas, and if A is Harrop, then $A \supset \exists x B(x)$ is provable in intuitionistic arithmetic iff $\exists x(A \supset B(x))$ is provable in intuitionistic arithmetic. This generalized the following result of Kreisel[1958]: if A,B lack occurences of the positive connectives "$\vee$" and "$\exists$", and if A, $\exists x B(x)$ are closed, then - again - $A \supset \exists x B(x)$ is provable in intuitionistic arithmetic iff $\exists x(A \supset B(x))$ is provable in the same system. As it happens, the examples presented in chapter 4 effectively rely only on Kreisel's result and not on Harrop's generalization, since all axioms used are intuitionistically equivalent to formulas in which neither "$\vee$" nor "$\exists$" appear.)

The following properties of normal proofs make it possible to use normalization to "run" proofs.

(1) Since each of the reduction rules preserves the end-formula of the proof to which it is applied, the end-formula of the normal form of a proof will always be the same as the end-formula of the original proof.

(2) A normal, Harrop proof of an existential formula $\exists x A(x)$ has the form:

$$\begin{array}{c} \Pi \\ A(t) \\ \hline \exists x A(x) \end{array} \; \exists I$$

Thus, a normal, Harrop proof of the existence of an object with a certain property contains a proof that a particular object has that property, and a term denoting that object can be easily extracted from the proof.

(3) A normal, Harrop proof of a formula of the form $A \vee B$ has one of the following forms:

$$\begin{array}{c} \Pi \\ A \\ \hline A \vee B \end{array} \qquad \begin{array}{c} \Pi \\ B \\ \hline A \vee B \end{array}$$

Now, it is evident that normalization allows one to pass mechanically from a Harrop proof of $\forall x \exists y A(x,y)$ and a term t_1 to a term t_2 together with a proof of $A(t_1,t_2)$. To do this, one simply applies the theorem $\forall x \exists y A(x,y)$ to the value t_1 (by use of the $\forall$ - elimination rule), and normalizes the resulting proof. By (2) above, the output value t_2 can be extracted from the next to last step of the normal proof. Similarly, a closed Harrop proof of $\forall x(A(x) \vee B(x))$ provides a uniform way of deciding which of A,B holds for any particular value of x.

2.4 Proof normalization

The reduction rules used in Prawitz's normalization procedure for natural deduction proofs are given below. The rules may be applied at any position in a proof tree. That is to say, any piece of the proof tree which matches the template given on the left hand side of a rule may be replaced by the appropriate instantiation of the right hand side of the rule, and this replacement constitutes an application of the rule. Notice that each rule removes a configuration in which an introduction rule is followed immediately by an elimination rule.

The following notation is used: $\Pi[\underline{x}\leftarrow\underline{t}]$ denotes the result of replacing all free occurences of the variables $\underline{x}$ by the terms $\underline{t}$ in the proof Π. The figure

$$\begin{matrix}\Pi\\ A\end{matrix}$$

denotes a proof P which has A as its end-formula. The figure

$$\begin{matrix}\Pi_1\\ [A]\\ \Pi_2\end{matrix}$$

denotes the result of replacing each open occurence of the assumption A by the proof Π_2 which has A as its end-formula. In both the substitution of terms for variables, and the substitution of proofs for assumptions, it may be necessary to change the names of variables bound by the $\forall$-introduction and $\exists$-elimination inferences; in this respect, substitution into proofs is similar to substitution into formulas or into λ-expressions.

$\wedge$-reduction:

$$\wedge E\dfrac{\wedge I\dfrac{\begin{matrix}\Pi_1 & \Pi_2\\ A & B\end{matrix}}{A\wedge B}}{A} \qquad \Rightarrow \qquad \begin{matrix}\Pi_1\\ A\end{matrix}$$

$$\wedge E\dfrac{\wedge I\dfrac{\begin{matrix}\Pi_1 & \Pi_2\\ A & B\end{matrix}}{A\wedge B}}{B} \qquad \Rightarrow \qquad \begin{matrix}\Pi_2\\ B\end{matrix}$$

$\vee$-reduction:

$$\vee\text{E}\,\dfrac{\vee\text{I}\,\dfrac{\begin{matrix}\Pi_1\\ A\end{matrix}}{A\vee B}\qquad\begin{matrix}[A]\\ \Pi_2\\ C\end{matrix}\qquad\begin{matrix}[B]\\ \Pi_3\\ C\end{matrix}}{C}\qquad\Rightarrow\qquad\begin{matrix}\Pi_1\\ [A]\\ \Pi_2\\ C\end{matrix}$$

$$\vee\text{E}\,\dfrac{\vee\text{I}\,\dfrac{\begin{matrix}\Pi_1\\ B\end{matrix}}{A\vee B}\qquad\begin{matrix}[A]\\ \Pi_2\\ C\end{matrix}\qquad\begin{matrix}[B]\\ \Pi_3\\ C\end{matrix}}{C}\qquad\Rightarrow\qquad\begin{matrix}\Pi_1\\ [B]\\ \Pi_3\\ C\end{matrix}$$

$\supset$-reduction:

$$\supset\text{E}\,\dfrac{\begin{matrix}\Pi_1\\ A\end{matrix}\qquad\supset\text{I}\,\dfrac{\begin{matrix}(A)\\ \Pi_2\\ B\end{matrix}}{A\supset B}}{B}\qquad\Rightarrow\qquad\begin{matrix}\Pi_1\\ [A]\\ \Pi_2\\ B\end{matrix}$$

$\forall$-reduction:

$$\forall\text{E}\,\dfrac{\forall\text{I}\,\dfrac{\begin{matrix}\Pi\\ A\end{matrix}}{\forall\underline{x}A}}{A[\underline{x}\leftarrow\underline{t}]}\qquad\Rightarrow\qquad\begin{matrix}\Pi[\underline{x}\leftarrow\underline{t}]\\ A[\underline{x}\leftarrow\underline{t}]\end{matrix}$$

$\exists$-reduction:

$$\exists\text{E}\,\dfrac{\exists\text{I}\,\dfrac{\begin{matrix}\Pi_1\\ A[\underline{x}\leftarrow\underline{t}]\end{matrix}}{\exists x A}\qquad\begin{matrix}[A]\\ \Pi_2\\ C\end{matrix}}{C}\qquad\Rightarrow\qquad\begin{matrix}\Pi_1\\ A[\underline{x}\leftarrow\underline{t}]\\ \Pi_2[\underline{x}\leftarrow\underline{t}]\\ C\end{matrix}$$

The reduction system given by the above reduction rules has the strong normalization property(Prawitz[1969]). We have left out the permutation rules, because they are not necessary for the execution of proofs.

2.5 Proof procedures

Let $A = \forall \underline{x} \varphi(\underline{x})$ be a closed non-Harrop formula. Suppose that one has a mechanical procedure γ which, when given a vector of closed terms $\underline{t}$, supplies a closed Harrop proof $\gamma(\underline{t})$ of the formula $\varphi(\underline{t})$. Such a procedure will be called a "proof procedure for A". It turns out that the availability of such a proof procedure makes it possible to execute proofs in which A is stated as a lemma. That is to say, it is not necessary for the purposes of proof execution to have a particular closed Harrop proof of a non-Harrop universal formula $\forall \underline{x} \varphi(\underline{x})$; it is sufficient to have a method for generating closed Harrop proofs of each closed instance $\varphi(\underline{t})$ of φ.

We require a proof procedure γ for $\forall \underline{x} \varphi(\underline{x})$ to supply a proof of $\varphi(\underline{t})$ only under the condition that $\underline{t}$ is closed. Nonetheless, it is convenient to *allow* a proof procedure to supply (not necessarily closed) proofs of $\varphi(\underline{t})$ for *some* vectors $\underline{t}$ of terms which are not closed, depending on circumstances. Thus we formally define a *proof procedure for* $\forall \underline{x} \varphi(\underline{x})$ to be a mechanical procedure γ with the following properties. (1) When γ is applied to any vector of terms $\underline{t}$, it returns either the atomic message "FAIL", or a Harrop proof of $\varphi(\underline{t})$. (2) If $\underline{t}$ is composed of closed terms, then $\gamma(\underline{t})$ must be a *closed* proof, and not the message "FAIL".

The use of proof procedures may be integrated into the normalization process by adding the following rule to the class of reduction rules for proofs given above.

lemma-reduction:

$$\forall E \frac{\text{lemma}: \forall \underline{x} \varphi}{\varphi(\underline{t})} \qquad \Rightarrow \qquad \gamma(\underline{t})$$

condition: γ is the proof procedure for $\forall \underline{x} \varphi$,
and $\gamma(\underline{t}) \neq \text{FAIL}$

We shall henceforth use the word "lemma" as a technical term which denotes a universal formula for which a proof procedure has been supplied. The set of lemmas together with their associated proof procedures is - like the language L - a parameter of the definition of the class of proofs, and of the class of normalization reductions. We assume that the proofs generated by proof procedures do not themselves make use of lemmas.

The addition of lemma-reduction to the set of reduction rules does not interfere with the strong normalization property. Also, the various properties of normal proofs which were given in section 2.3 continue to hold if we add the restriction that the normal proofs in question be *closed.* Since reductions on proofs pass from closed proofs to closed proofs (section 2.9), it follows that closed proofs continue to have all of the computationally useful properties remarked on in section 2.3.

In the example to be given in section 2.8, only one lemma is used, namely the lemma ∀x y $(x \leq y \vee x > y)$ which states the decidability of numerical inequalities. The proof procedure for this lemma simply provides the proof

$$\vee I \frac{t_1 \leq t_2}{t_1 \leq t_2 \vee t_1 > t_2}$$

if t_1 and t_2 are closed and the formula $t_1 \leq t_2$ is true, and the proof

$$\vee I \frac{t_1 > t_2}{t_1 \leq t_2 \vee t_1 > t_2}$$

if t_1 and t_2 are closed and the formula $t_1 > t_2$ is true; if t_1 or t_2 contains a free variable, then "FAIL" is returned. Proof procedures for formulas of the form $\forall \underline{x}(R(\underline{x}) \vee \neg R(\underline{x}))$ with R atomic play a role in normalization which corresponds to the role played by primitive predicates in programming languages.

2.6 Reductions on terms of L

Suppose that one has a reduction system <T,R> for the terms of a first order language L. Then the reductions R can be incorporated into proof normalization simply by by allowing them to be applied at will to terms which appear in the formulas of proofs. In such a hybrid reduction system there is little interaction between the reductions on terms and the reductions on proofs. If both the reduction system for terms and the reduction system for proofs have the termination property, then so will the hybrid reduction system. This holds for the uniqueness property as well, so long as the proof procedures for non-Harrop formulas commute with term reductions.

As an example, consider a formulation of first order arithmetic in which terms are built up from variables, decimal (or binary) notations for natural numbers, and function symbols for successor, addition and multiplication. Consider also the reduction system consisting of the single rule which replaces closed numerical terms by decimal notations for their values. In computing numerical functions by means of proof normalization, the use of this term reduction rule allows the addition and multiplication of numbers to be carried out by efficient machinery external to the normalization procedure. In particular, the rule can be implemented in such a way as to take advantage of the arithmetic hardware possessed by most computers.

Reductions on terms will receive little explicit attention in the rest of this thesis. However, the presence of a well-behaved reduction system for terms will not affect any of the results about proof normalization presented in this chapter or in chapter 3. By "well-behaved", we mean (1) terminating, and (2) value-preserving with respect to the model (if any) currently under consideration Whenever reductions on terms are mentioned, the reader is to assume that properties (1) and (2) hold.

2.7 Pruning

The pruning operations are as follows.

$$\vee E\;\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3\\ A\vee B & C & C\end{matrix}}{C} \quad\Rightarrow\quad \begin{matrix}\Pi_2\\ C\end{matrix}$$

if A does not appear as an open assumption in Π_2.

$$\vee E\;\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3\\ A\vee B & C & C\end{matrix}}{C} \quad\Rightarrow\quad \begin{matrix}\Pi_3\\ C\end{matrix}$$

if B does not appear as an open assumption in Π_3.

There is also a pruning operation:

$$\exists E\;\frac{\begin{matrix}\Pi_1 & \Pi_2\\ \exists x A & C\end{matrix}}{C} \quad\Rightarrow\quad \begin{matrix}\Pi_2\\ C\end{matrix}$$

if A does not appear as an open assumptioon in Π_2

for the $\exists$-elimination inference, but it will play no role in the work described in this thesis. Henceforth when we speak of a "pruning operation" we mean one of the two pruning operations for $\vee$-elimination.

It should now be clear why the pruning operations are unlikely to be useful when applied to proofs as originally given by people. The inferences removed by pruning are redundant, and one does not expect to find them in proofs which have been constructed in a purposeful way. However, the example given in the next section demonstrates that proofs which result from simple automatic processes may contain such redundancies; in particular the process of specializing a proof by normalization may introduce redundancies where none had at first appeared.

The pruning operations may be adjoined to the set of reductions used in normalization. The resulting reduction system retains the termination property, although the uniqueness property is lost. This loss of uniqueness is an advantage and not a defect of pruning. Pruning allows us to reduce proofs to a variety of equally satisfactory normal forms, some of which can be arrived at more quickly than the normal form which results from normalization without pruning. Thus, by dropping the uniqueneess requirement, we gain efficiency.

2.8 An example

The simplest algorithms to which the pruning operations are usefully applicable are pure case analysis algorithms - algorithms which can be expressed by "plain" conditional expressions. In what follows, we present a very small case analysis algorithm which is nonetheless sufficient to illustrate the main points which we wish to make about pruning. These points are: (1) pruning may be used to increase the efficiency of specializations of algorithms, and (2) conventional descriptions of algorithms do not contain the data necessary for the improvements in efficiency realized by pruning. Consider, then, the following algorithm - given as a conditional expression - for computing an upper bound for both the sum and the product of two positive rational numbers x and y:

$\mathbf{u}(x,y) = \text{if } x \leq 1 \text{ then } y+1 \text{ else } (\text{if } y \leq 1 \text{ then } x+1 \text{ else } 2xy)$

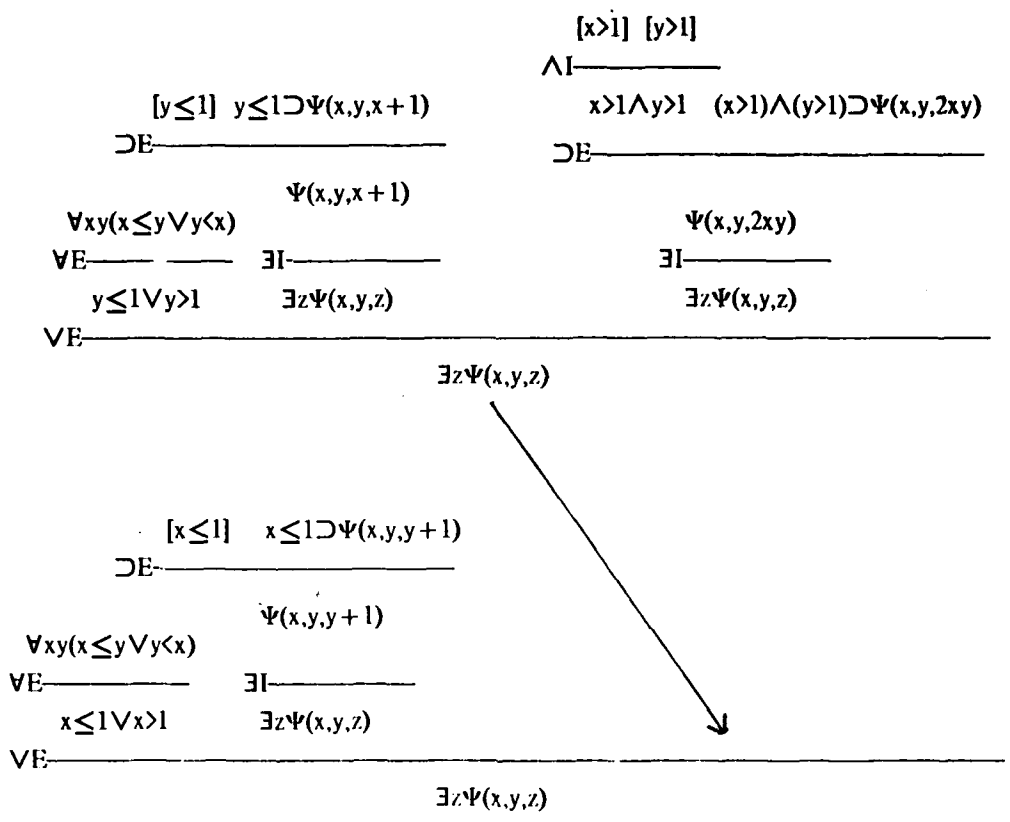
We will use the bold faced letter **u** to refer both to the algorithm, considered as an abstract method which can be formalized in various ways, and to the above concrete conditional term.

Now, suppose that the value .5 is given for y in advance, and that we wish to optimize **u** given this additional information. The best we can do, if supplied only with the conditional expression as a description of the algorithm, is to symbolically execute the expression on the arguments x, .5. The result is:

$\mathbf{u}(x,.5) = \text{if } x \leq 1 \text{ then } 1.5 \text{ else } x+1$

As will be seen below, the formalization of this upper bound algorithm as a proof allows u(x,.5) to be automatically simplified, by use of normalization and pruning, to the expression $x+1$. The fact that $x+1$ is an upper bound for both $x+.5$ and $.5x$ does not depend on x being less than or equal to one; this dependency information is contained in the proof, and allows the automatic removal of the unnecessary case split according to the size of x. Note that the pruning optimization has the unusual quality that it modifies the function computed by the expression to which it is applied. However, pruning is guaranteed to preserve the validity of an algorithm for the specification embodied in the end-formula of the proof describing the algorithm. Also note that no transformation on conventional computational descriptions can have the same effect as pruning. Conventional descriptions contain information only about the function to be computed, and not about the purpose of the computation, and therefore valid transformations on such descriptions must - unlike pruning - preserve extensional meaning.

The following natural deduction proof formalizes the upper bound algorithm **u**. In the proof and elsewhere $\Psi(x,y,z)$ is used to abbreviate the formula $(z \geq x+y) \wedge (z \geq xy)$. Leaves of the proof tree which are not surrounded by brackets designate axioms or lemmas. Three Harrop axioms ("$x \leq 1 \supset \Psi(x,y,y+1)$", "$y \leq 1 \supset \Psi(x,y,x+1)$", and "$(x>1) \wedge (y>1) \supset \Psi(x,y,2xy)$") and one lemma $\forall x\, y(x \leq y \vee y < x)$, appear in the proof. We assume that the proof procedure described in section 2.5 above has been provided for the lemma. Also, reduction rules for numerical terms, which will, for example, reduce 2+1 to 3, are assumed to be present. (The details of the notation used for rational numbers and of the reductions which apply to numerical terms are unimportant for the purposes of the current discussion.) We will use the capital letter U to designate the proof.



Note that we have neglected to universally quantify the variables x,y so as to arrive at a proof in the standard ∀∃ form. In the current simple context it is more convenient for purposes of exposition to leave the quantification implicit, and to specify that input values to the proof viewed as an algorithm be substituted for the free variables. More precisely, in order to compute an upper bound for the sum and product of two input values v_1 and v_2 by means of normalization, v_1 and v_2 are first substituted for x,y throughout the proof U, and then the proof is normalized.

Normalization of simple case analysis proofs such as U makes use only of the ∨-reduction rules (section 2.2) and perhaps of proof procedures for lemmas. In this restricted case, normalization of proofs corresponds closely to the execution of conditional terms by means of repeated applications of the reduction rules:

C_1: (if TRUE then t_1 else t_2) $\Rightarrow$ t_1

C_2: (if FALSE then t_1 else t_2) $\Rightarrow$ t_2

A_1: $R(t_1,t_2 \ldots t_n) \Rightarrow$ TRUE if R is an atomic relation, $t_1,t_2 \ldots t_n$ are closed ground terms, and $R(t_1,t_2 \ldots t_n)$ holds

A_2: $R(t_1,t_2 \ldots t_n) \Rightarrow$ FALSE if R is an atomic relation, $t_1,t_2 \ldots t_n$ are closed ground terms, and $R(t_1,t_2 \ldots t_n)$ does not hold

The two V-reduction rules correspond in their effect to C_1 and C_2, while proof-procedures for lemmas of the form $\forall x_1,x_2 \ldots x_n(R(x_1,x_2 \ldots x_n) \vee \neg R(x_1,x_2 \ldots x_n))$ correspond to the rules A_1 and A_2.

More specifically, the V-reduction rule takes an V-elimination inference

$$\vee E\frac{\begin{matrix}\Pi_1 \\ A \vee B\end{matrix} \quad \begin{matrix}\Pi_2 \\ C\end{matrix} \quad \begin{matrix}\Pi_3 \\ C\end{matrix}}{C}$$

in which the proof Π_1 of the first premise indicates which one of A and B is true; depending on whether it is A or B that holds, either the second "branch" Π_2 or the third "branch" Π_3 of the inference is selected. This corresponds to making use of a binary decision between TRUE and FALSE in a conditional expression to select a branch of the conditional.

As an example, the reader may wish to carry out the normalization of U when inputs 2 and .5 are substituted for x and y, respectively. The normalization of the proof will parallel the normalization of the term

if $2 \leq 1$ then $.5+1$ else (if $.5 \leq 1$ then $2+1$ else $2(2)(.5)$)

with respect to the reduction rules C_1,C_2,A_1,A_2 given above. The final result of the normalization will be:

$$\exists I\frac{\supset E\dfrac{.5<1 \qquad .5 \leq 1 \supset \Psi(2,\ .5,\ 3)}{\Psi(2,\ .5,\ 3)}}{\exists z \Psi(2,\ .5,\ z)}$$

The value returned by this proof is "3".

In order to specialize the algorithm expressed by U to the case where y is fixed at .5, .5 is substituted for y throughout the proof, and the result is normalized. This process yields the following "specialized" proof:

$$
\vee E\,\dfrac{\forall E\,\dfrac{\forall xy(x\leq y \vee y<x)}{x\leq 1 \vee x>1} \qquad \exists I\,\dfrac{\supset E\,\dfrac{[x\leq 1] \qquad x\leq 1\supset\Psi(x,\ .5,\ 1.5)}{\Psi(x,\ .5,\ 1.5)}}{\exists z\Psi(x,\ .5,\ z)} \qquad \exists I\,\dfrac{\supset E\,\dfrac{.5\leq 1 \qquad .5\leq 1\supset\Psi(x,\ .5,\ x+1)}{\Psi(x,\ .5,\ x+1)}}{\exists z\Psi(x,\ .5,\ z)}}{\exists z\Psi(x,\ .5,\ z)}
$$

This proof corresponds to the specialized conditional term, "if x$\leq$1 then 1.5 else x+1". A further optimization is applicable to the specialized proof which is not applicable to the conditional term, namely pruning. The second minor premise of the $\vee$-elimination inference in the specialized proof above does not depend on the assumption x>1. It is this fact about the dependency structure of the computation that the proof U, but not the conditional term **u**, formalizes, and which allows pruning to take place. The result of applying pruning is:

$$
\exists I\,\dfrac{\supset E\,\dfrac{.5\leq 1 \qquad .5\leq 1\supset\Psi(x,.5,x+1)}{\Psi(x,.5,x+1)}}{\exists z\Psi(x,.5,z)}
$$

This represents the same algorithm as the conditional term "x+1".

Note that, if comparison is a very cheap operation, and adding is very expensive, then it might happen that "x+1" has an average case efficiency which is worse than "if x$\leq$1 then 1.5 else x+1". This illustrates the general point that pruning is not *guaranteed* to increase efficiency. However, pruning often improves the efficiency of an algorithm, and *always* reduces its size. (Size reduction is an important effect of pruning in the experiments on bin-packing; see chapter 4)

2.9 Summary: conditions for the computational usefulness of proofs

In what follows, we collect together the various results and conditions which are relevant to the usefulness of proofs for computation, and explicitly describe the relationships between them. First of all, we have the results about the reduction rules involved in normalization:

(1a) Syntactic validity of the reduction rules for proofs (given in section 2.4) and of pruning: each of these operations yields a well-formed proof when applied to a well-formed proof.

(1b) Preservation of the end-formula: the reduction rules do not modify the end-formula of a proof.

(1c) Termination: every sequence of applications of reduction rules to a proof terminates.

(1d) Preservation of "closedness": a reduction rule yields a closed proof when applied to a closed proof.

Second there is the result concerning the normal form (sections 2.3, 2.5):

(2) A normal, closed, Harrop proof of $\exists x A$ has the form,

$$\exists I \frac{\begin{matrix}\Pi \\ A(t)\end{matrix}}{\exists x A(x)}$$

All of the above results are purely syntactic in nature. No mention is made of the meaning of the formulas which appear in proofs. However, we have,

(3) *The inference rules of natural deduction are sound with respect to the usual Tarskian semantics.*

The inference rules are also sound for the intuitionistic notion of validity. As a consequence, each of the remarks made below will continue to hold if the words truth and validity are taken to refer to the intuitionistic rather than the classical notions.

The final result which guarantees the possibility of executing proofs of $\forall\exists$ formulas is:

(4) If Π is a proof of $\exists x A(x)$ meeting certain conditions, then the normalization procedure terminates when applied to Π, and results in a proof having the form,

$$\exists I \frac{\begin{matrix}\Pi \\ A(t)\end{matrix}}{\exists x A(x)}$$

where $A(t)$ is *true* (in some intended model).

The conditions for the result (4) are: (a) the proof must be closed, (b) all axioms appearing in the proof must be Harrop formulas, (c) all axioms appearing in the proof must be true, and (d) the axioms which appear in proofs generated by proof procedures must be true.

The proof of result (4) from the various results under (1), (2), (3) above is as follows: If Π is a proof of $\exists x A(x)$ meeting the conditions (a)-(d) of (4), then

° normalization terminates on Π by (1c),

° and yields a proof in the form,

$$\exists I \frac{\begin{array}{c}\Pi \\ A(t)\end{array}}{\exists x A(x)}$$

by conditions (a),(b) and results (1a),(1b),(1d),(2);

° finally A(t) is true by result (3) and conditions (c) anc (d).

We wish to emphasize the degree to which the various results and conditions which come into the proof of (4) are independent. In particular, none of the results under (1) and (2) depends in any way on the truth of the axioms which appear in proofs. Thus syntactic and semantic considerations do not interact and can be examined separately.

Chapter 3

Efficient Implementation of Operations on Proofs

The normalization and pruning operations described in the last chapter are quite inefficient if implemented in a literal minded way. The problem is not so much that the *asymptotic* efficiency of an algorithm is degraded if it is formalized as a proof, but rather that the elementary operations which are used in normalization are computationally expensive. For example, the substitution of a proof for occurences of an assumption is an expensive operation, both in time and space.

However, as we will show in this chapter, normalization and pruning can be implemented in an efficient manner if an appropriate data structure for proofs is used. Specifically, we will represent natural deduction proofs by terms of an extended λ-calculus. The normalization of such λ-caluclus terms can be implemented efficiently by using environments instead of literal substitutions, as is done in interpreters for λ-calculus based languages such as LISP.

In section 3.1, we describe the connection between the natural deduction formalism and the typed λ-calculus. Emphasis is placed on pure implicational logic, where the connection is most direct. In sections 3.2 - 3.4 we present a λ-calculus based representation for natural deduction proofs of full predicate logic. Sections 3.5 and 3.6 concern the manner in which normalization and pruning operations apply to this representation. In section 3.7 we describe an additional reduction rule used in the experiments of chapter 4 - namely, the permutation rule for ∨-elimination. In section 3.8, schematic examples are presented which illustrate the effect that pruning can have on the computational efficiency of proofs.

3.1 Natural deduction and the typed λ-calculus

The close structural correspondence between natural deduction proofs and terms of the typed λ-calculus has been known for some time, and forms the basis for the calculi of constructions developed by Scott[1970], Howard[1980], DeBrujin[1970], Martin-Löf[1979], and others. (The calculus which is closest to our own "p-calculus" [section 3.2] is Martin-Löf's[1979] theory of types.) The central idea here is that the same elementary operations may be used in (1) constructing and applying general methods of computation, and in (2) establishing and applying general truths. As an example, consider (a) a term t(x) of the typed λ-calculus in which (only) the variable x appears free, (b) a proof Π of a formula B in which (only) the formula A appears as an open assumption. In both the cases (a) and (b), one has an incompletely given construct; t does not denote any particular object, but will do so once a concrete value has been supplied for x and substituted into t; similarly, Π does not establish the truth of B, but will do so when any proof of A is given and substituted for occurences of the assumption. Thus, in both cases, the incomplete construct in question supplies a general method for passing from a *value* (for the variable x or the assumption A) to a *result* (of substitution). One may apply the operation of *abstraction* to the incomplete construct so as to arrive at a term or proof which describes this *general method.* In case (a) the abstraction is written, "$\lambda x.t$', while in case (b) the abstraction is the proof,

$$\supset\! I \, \dfrac{\begin{matrix}[A]\\ \Pi\\ B\end{matrix}}{A \supset B}$$

One also has the converse operation at one's disposal, namely, *application.* If one has a term t_1 which describes a general method, and a term t_2 of the appropriate type, then one may form the term "$t_1(t_2)$", which denotes the result of applying the general method t_1 to the input t_2. Similarly, if one has a proof Π_1 of $A \supset B$ - that is to say, a general method for getting from proofs of A to proofs of B - and also a particular proof Π_2 of A, then one may form a proof which denotes the result of applying Π_1 to Π_2. That proof is:

$$\supset\! E \, \dfrac{\begin{matrix}\Pi_2\\ A\end{matrix} \qquad \begin{matrix}\Pi_1\\ A \supset B\end{matrix}}{B}$$

Thus, the constructor "λ" which is used in building up λ-terms corresponds to the inference rule ⊃I, while the constructor for application: $t_1(t_2)$ corresponds to the inference rule ⊃E.

In both the λ-calculus and the formalism of natural deduction proofs, normalization involves applying general methods (as described by abstractions) to given inputs. Specifically, the *β*-conversion rule for the λ-calculus reduces an application $(\lambda x.t_1)(t_2)$ of an abstraction $(\lambda x.t_1)$ to an input t_2 to the term $t_1[x \leftarrow t_2]$. The corresponding reduction for proofs is just implication reduction:

$$\supset\! E \, \dfrac{\begin{matrix}\Pi_1\\ A\end{matrix} \qquad \supset\! I \, \dfrac{\begin{matrix}[A]\\ \Pi_2\\ B\end{matrix}}{A \supset B}}{B} \qquad \Rightarrow \qquad \begin{matrix}\Pi_1\\ [A]\\ \Pi_2\\ B\end{matrix}$$

For natural deduction proofs of pure implicational logic, the correspondence to the typed λ-calculus is exact; any such proof may rewritten as a λ-calculus term by (1) replacing assumptions by variables, (2) replacing each ⊃-introduction inference by a λ-abstraction of the variable corresponding to the assumption discharged by the inference, and (3) replacing ⊃-eliminations by applications. This change of notation from proof to λ-caluclus language results in no loss of information, and furthermore, the ⊃-reduction operation on proofs is thereby mapped directly onto the β-conversion operation on λ-calculus terms. The particulars of this "change of notation" are as follows.

First we present a formal definition of the typed λ-calculus. We start with a collection of symbols $\tau_1, \ldots \tau_n$ called the "base types". Complex types are built up from the base types $\tau_1, \ldots \tau_n$ by the binary constructor "$\rightarrow$"; the inductive definition is: (1) each τ_i is a type; (2) if τ, ρ are types then so is "$\tau \rightarrow \rho$" . The base types are intended to denote sets of "primitive" objects, while $\tau \rightarrow \rho$ is intended to denote the set of mappings from objects of type τ to objects of type ρ. Next, we assume that an infinite set V_τ of variables is given for each type τ. The elements of V_τ are called "variables of type τ". V_τ and V_ρ are assumed to be disjoint for distinct types τ and ρ. The following inductive clauses define the notion of a term of type τ.

(1) each variable v_τ of type τ is a term of type τ.

(2) If t is of type τ and x is a variable of type ρ then $\lambda x.t$ is a term of type $\rho \rightarrow \tau$.

(3) If t_1 is of type $\tau \rightarrow \rho$ and t_2 is of type τ, then $t_1(t_2)$ is a term of type ρ.

By "pure implicational logic" is meant the restricted natural deduction system in which formulas are built up from propositional constants by use of implication alone, and in whose proofs only the ⊃E and ⊃I inferences appear. The formulas which appear in proofs correspond to the types of λ-terms; a propositional constant P corresponds to a base type τ_P, while a formula A ⊃ B corresponds to a type $\tau_A \rightarrow \tau_B$. More precisely, we assign to each implicational formula A a type τ_A according to the following rules. (1) Each propositional constant P is assigned a base type τ_P. (2) If the formula A has been assigned the type τ_A, and the formula B has been assigned the type τ_B, then the formula A ⊃ B is assigned the type $\tau_A \rightarrow \tau_B$.

We now define the map Γ which rewrites proofs as λ-terms. It is assumed to start with that variables of appropriate types have been selected for labeling formulas; we assume, that is to say, that a unique variable v_A of type τ_A has been assigned to each formula A. Γ is defined by induction on the structure of proofs. We use the notation Γ: Π ⇒ t to indicate that the value of Γ applied to Π is t.

(1) Base case: Γ: $[A] \Rightarrow v_A$

(That is, Γ when applied to a proof which consists simply of an assumption $[A]$ yields the variable v_A which labels the formula A.)

(2)

$$\Gamma: \qquad \supset I \frac{\begin{array}{c}[A]\\ \Pi \\ B\end{array}}{A \supset B} \qquad \Rightarrow \qquad \lambda v_A.\, \Gamma(\Pi)$$

(3)

$$\Gamma: \qquad \supset E \frac{\begin{array}{cc}\Pi_1 & \Pi_2\\ A & A \supset B\end{array}}{B} \qquad \Rightarrow \qquad (\Gamma(\Pi_1))(\Gamma(\Pi_2))$$

For example, the proof

$$\supset I \frac{\supset I \dfrac{\supset E \dfrac{[A] \qquad \supset E \dfrac{[A] \qquad [A \supset (A \supset B)]}{A \supset B}}{B}}{A \supset B}}{(A \supset (A \supset B)) \supset (A \supset B)}$$

when written in λ-calculus notation yields the term

$$\lambda v_{[A \supset (A \supset B)]}.\ \lambda v_A.\ \{(v_{[A \supset (A \supset B)]}(v_A))(v_A)\}$$

of type $(\tau_A \rightarrow (\tau_A \rightarrow \tau_B)) \rightarrow (\tau_A \rightarrow \tau_B) \ = \ \tau_{[(A \supset (A \supset B)) \supset (A \supset B)]}$.

Notice that, for any proof Π with endformula A, the type of the λ-caluclus notation $\Gamma(\Pi)$ for that proof is τ_A. Similarly, the types of the subterms of $\Gamma(\Pi)$ correspond to the endformulas of the subproofs from which those subterms arise.

What we have done so far is to show that natural deduction proofs of a restricted system can be represented as λ-calculus terms. It is possible to represent any natural deduction proof in the same style, under the condition that appropriate additional contructors are adjoined to the λ-calculus. But before going on to describe the λ-calculus formulation of full predicate calculus, it worthwhile looking more closely at the differences between the proof notation and the λ-calculus notation for proofs of pure implicational logic.

Both proofs and λ-terms may be regarded as labeled trees: proof trees are labeled by formulas and inference rule names, and "λ-trees" by variables (at leaves) and construction rule names (at interior nodes). From this point of view the difference between proof notation and λ-caluleus notation lies in the choice of information which is explicitly stored on the tree. In proofs, a formula is stored at every node. In a λ-term, the corresponding type information is associated only with the variables which appear at the leaves of the tree, and must be computed for other nodes. In proofs, the connection between inference rules and the sets of assumptions which they discharge must be derived from "type information" (ie formulas on the tree). In λ-terms, this information is represented more explicitly: the discharged assumptions are labeled by a bound variable.

Suppose that all type information is dropped from a λ-term - that the typed variables are replaced one for one by variables with which no type information is associated. Then the resulting untyped λ-term represents the "logical structure" of a proof, in the following sense. The underlying tree of the untyped term records a sequence of applications of inference rules (in λ-calculus notation), and also describes the graph of connections between inference rules and the assumptions (represented by variables) which they discharge. Thus if one were to take a proof tree, and strip off the formulas which appear on the tree, while retaining a record of the "logical structure" of the proof, then the result would contain the same information as an untyped λ-expression. (The logical structure of proofs in the current sense is exactly the structure preserved by isomorphisms between proofs in the sense of Statman[1974].)

Now, notice that the normalization reductions of the λ-caluclus make no use of type information; if one wishes to normalize a typed λ-caluleus term, one is free to throw away the types before doing the normalization, and the result will be no different. Correspondingly, the sequence of steps taken in the normalization of a proof depends only on the "logical structure" of the proof in the sense of the last paragraph. Two proof trees on which different formulas appear will be subjected to the same sequence of reduction steps by normalization, so long as the inference rules and the structure of discharges of assumptions on the two trees are the same.

When we consider λ-calculus notation for arbitrary natural deduction proofs, it will be seen that once again type information is not necessary for normalization. Furthermore, untyped terms contain the desired output of computations, and can be subjected to pruning.

That is to say, the "logical structure" of a proof as expressed by an untyped λ-term is sufficient not only to determine the form of the normalization sequence, but also to determine the output value which is extracted from a normal proof, and to allow pruning to take place. Thus, for practical purposes, it is always sufficient to deal with the untyped variants of proofs.

The following remarks summarize the interest of using a λ-calculus based notation for proofs.

(1) The untyped variant of the λ-calculus notation for a proof contains exactly that information which is relevant to the execution and pruning of the proof.

(2) An efficient technology exists for normalization of (proofs expressed as) λ-calculus terms.

3.2 The p-calculus

In order to arrive at a notation of the kind discussed in the last section which is adequate for arbitrary natural deduction proofs, new constructors for the inference rules other than ⊃-introduction and ⊃-elimination are added to the λ-calculus, namely : (1) pairing (for ∧-introduction), (2) unpairing (for ∧-elimination), (3) OI_1 and OI_2 (for ∨-introduction), (3) OE (for ∨-elimination), (4) EI (for ∃-introduction), and (5) EE (for ∃-elimination). ∀-introduction and ∀-elimination are treated using the "old" constructors λ-abstraction and application. The extended system just described will be referred to as the "p-calculus".

We will have occasion to deal with both a typed and an untyped variant of the p-calculus. The relationship between proofs, typed terms, and untyped terms is the same for the p-calculus as it is for the "plain" λ-calculus. Namely, a typed term of the p-calculus constitutes a complete representation of a proof, while an untyped term serves to express only that information in a proof which is needed for execution and pruning.

The "types" which will be assigned to terms of the typed p-calculus will not be types in the ordinary sense; rather, they will be formulas of first order logic. The connection between formulas and types given in the last section for implicational logic can be extended to the p-calculus treatment of full first order logic; it is possible to assign types of the ordinary kind (ie classes of functions) to arbitrary first order formulas, and to assign functions to p-calculus terms, in such a way that the two assignments are consistent. Specifically, a term of "type" φ will denote a function which actually belongs to the type assigned to φ. However, none of the results which will concern us here depend on the details of such assignments, or indeed on such assignments being possible at all. The reader will find further information on formulas as types in Scott[1970], and Howard[1980].

We define the untyped variant of the p-calculus as follows. The starting point for the definition is (1) an infinite set V of variables, (2) a first order language L, as described in section 2.1, (3) the special symbol #, and (4) a set D of "defined symbols" with associated arities. It is assumed that the variables V and the variables of L are distinct. The variables of L are called "object variables", while the variables in V are called "proof variables". The defined symbols D will be used as labels of proof procedures for lemmas, and in recursive definitions (section 3.4) as well. The letters "α, β", "f, g, h" and "x, y, z" will be used to designate proof variables, defined symbols, and object variables, respectively. The p-calculus P_L over L, then, is defined by the following inductive clauses. The phrase "p-term" is taken to designate an element of P_L.

(1) The terms and atomic formulas of L are p-terms (see section 2.1).

(2) The proof variables V are p-terms.

(3) The special constant # is a p-term.

(4) The defined symbols D are p-terms.

(4) If t_1,t_2 are p-terms, then so is $\langle t_1,t_2\rangle$ [pairing].

(5) If t is a p-terms then so are $\pi_1(t)$, $\pi_2(t)$ [unpairing].

(6) If α is a variable, and t is a p-term, then $\lambda\alpha.t$ is a p-term. [proof-abstraction]

(7) If $x_1\ x_2\ .\ .\ .\ x_n$ are variables, and t is a p-term, then $\lambda x_1\ x_2\ .\ .\ .\ x_n.t$ is a p-term. [object-abstraction]

(8) If t_1,t_2 are p-terms then so is $t_1(t_2)$. [Application]

(9) If α is a proof variable, and t_1,t_2,t_3 are p-terms then so is $OE(\alpha,t_1,t_2,t_3)$

(10) If α is a proof variable, x an object variable, and t_1,t_2 are p-terms, then $EE(x,\alpha,t_1,t_2)$ is a p-term.

Note that, in the case of object variables, we have chosen to introduce λ-abstraction of arbitrary arity as a primitive constructor rather than using "Currying". This simplifies the correspondence between λ-abstraction and ∀-introduction. Note that $\lambda x_1\ x_2 \ldots x_n.t$ is *not* an abbreviation for $\lambda x_1\ \lambda x_2. \ldots \lambda x_n.t$.

The above definition is given in terms of ordinary syntax suitable for written presentation. However, in our discussions of formal operations on terms, we will treat p-calculus terms as labeled trees, as was done in the discussion of the λ-calculus in the last section.

There are several ways in which one can go about representing terms by labeled trees, and the details of how this is done are not of any fundamental importance. However, in order to avoid confusion later, it is worthwhile deciding here on a specific representation. That representation is as follows. The relationship between a term and its immediate subterms is coded directly in the structure of the tree - each node represents a term, and the sons of the node represent the immediate subterms of that term. Leaf nodes are labeled by atomic symbols - proof variables, #, and symbols of L. Each non-leaf node is labeled by the constructor used for arriving at the current term from its immediate subterms, and by the variables which are bound by that constructor. The constructor which appears at the root node of any term is referred to as the "main constructor" of that term. The constructors are: PAIR, APPLY, π_1, π_2, OI_1, OI_2, OE, λ, EI, EE. In the typed variant of the p-calculus, nodes may be labeled by formulas as well. *Note that the variables bound by a constructor* - for example the "x" in λx.t or the "α" and "x" in $EE(x,\alpha,t_1,t_2)$ - are *not* regarded as subterms, but as a part of the information with which nodes of the tree are labeled.

In what follows, the notation "A(B)" for application is used in three different ways. (1) When A and B are p-calculus terms, A(B) denotes the p-term whose main constructor is APPLY and whose immediate subterms are A and B. (2) When A is a constructor (such as π_1) and B is a p-term, then A(B) designates the *result* of applying the constructor to the term B, that is to say, A(B) designates the p-term whose main constructor is A and whose immediate subterm is B. (3) If A is an operation on p-terms and B is a p-term, then A(B) will denote the result of applying A to B. Thus the notation A(B) serves both as an external syntax for a formal p-term whose main constructor is APPLY, and to denote the "actual" application of an operation to an object. This is an ambiguity of the mention/use kind. However, in each of the cases (1)-(3) context is sufficient to resolve the ambiguity.

In defining the typed variant of the p-calculus, it is most convenient to proceed by assigning types (ie formulas) not to variables, but rather to the nodes of p-terms. In particular, a *typed p-term* is a p-term some of whose nodes have been labeled by formulas according to certain rules. The formula assigned to a given node represents the type of the subterm rooted at that node (or, in proof language, the end-formula of the subproof rooted at the node). We follow traditional terminology, and refer to a typed p-term as a *construction*. The words "term" and "p-term" will be used to denote *untyped* p-terms.

Before describing the rules by which constructions are to be built up, we need to define the notions of bound and free occurences of variables, and of substitution, as they apply to constructions. We use the phrase "labeled p-term" to refer to a p-term to whose nodes formulas have been assigned in an arbitrary manner (in constrast to a *typed p-term* or *construction*, whose labeling must follow certain rules).

Let t be an labeled p-term. An *occurence of a variable in t* is an occurence of the variable either as a leaf of the p-term, or an occurence of the variable in one of the formulas assigned to the nodes of t. The notion of a *bound occurence* of a variable in a labeled p-term is defined below. The definition follows standard lines, but includes new clauses for the constructors EE and OE. (The new clauses express the fact that OE and EE, like λ, $\forall$ and $\exists$, have the effect of binding variables.)

(1) Each occurence of the variable α in t_2 or t_3 (but *not* in t_1) is a bound occurence of α in the terms (a) $OE(\alpha,t_1,t_2,t_3)$, (b) $EE(x,\alpha,t_1,t_2)$, (c) $\lambda\alpha.t_2$.

(2) Each occurence of the variable x in t_2 is a bound occurence of x in (a) $EE(x,\alpha,t_1,t_2)$, and *in (b)* $\lambda \underline{y}.t_2$ *if* x *is among the variables* $\underline{y}$.

(2) Each occurence of the variable x in the formula φ is a bound occurence of x in (a) $\exists x\varphi$, and in (b) $\forall \underline{y}.\varphi$ if x is among the variables $\underline{y}$.

(4) If t_1 is a subterm of t_2, each occurence of a variable in t_1 which is *bound in t_1* is also *bound in t_2*.

Any variable occurence which is not specified as bound by the above three rules is a *free occurence* of the variable.

The elementary operations on terms - notably the renaming of bound variables and substitution - are defined in exactly the same way for the p-calculus (typed or untyped) as they are for the plain λ-calculus. One only has to take the new variable-binding constructors OE and EE into account in the obvious way.

For example, the definition of α-conversion (renaming of one bound variable) includes the following clause for OE: Suppose that the terms t_2', t_3' result from the terms t_2, t_3 by the replacement of all free occurences of the variable α by the variable β. Suppose further that β does not itself occur free in either t_2 or t_3. Then one may replace the term $OE(\alpha,t_1,t_2,t_3)$ by the term $OE(\alpha, t_1, t_2', t_3')$. The other clauses for α-conversion are the standard clause for λ, and two clauses for EE - one for renaming the object variable, and one for renaming the proof variable.

The operation of substitution may be defined as follows: in order to substitute the term t_2 for the variable x in t_1, first rename all bound variables which appear in t_1 in such a way that no free variable of t_2 has a bound occurence in t_1. (This can be done by a series of α-conversions.) Then replace all free occurences of x in t_1 by t_2. The result of this operation will be denoted by, "$t_1[x \leftarrow t_2]$". Evidently the above definition does not fully specify the term which results from substitution because it leaves open the particular choice of variables which are used in renaming. However, the result *is* uniquely defined modulo renaming of bound variables. We shall henceforth regard as identical terms which differ only in the names of their bound variables (ie, terms which can be transformed into each other by means of α-conversions).

The notation $t_1[t_2 \leadsto t_3]$ designates the result of substituting t_2 for *some* occurences of t_3 in t_1. Whenever this notation is used, it is assumed that no bound variable of t_1 appears free in t_3. (Thus, no free variable of an occurence of t_3 within t_1 is bound by a constructor of t_1.) As in the case of substitution of terms for variables, the substitution of terms for terms involves changing of bound variable names in t_1 so as to avoid conflicts with the variables which appear free in t_2. Finally, $t_1[\underline{x} \leftarrow \underline{t}_2]$ denotes the result of substituting the terms $\underline{t}_2$ for *the variables* $\underline{x}$ *in parallel.*

We are now in a position to define the notion of a *typed p-term,* or *construction.* A construction is a labeled p-term which is built up according to the rules given below and which in addition satisfies the following general restrictions: (1) Every occurence of a proof variable in a construction t must be labeled by a formula. (2) Suppose that t' is any subterm of the construction t, and that α is a proof variable which occurs free in t'. Then every free occurence of α in t' must be labeled by the *same formula.*

The rules for building up constructions given below correspond exactly to the inference rules of natural deduction. The name of the inference rule corresponding to each rule is given in brackets next to the rule. We make use of the notation t:F to indicate a construction whose root is labeled by the formula F. (Other nodes of t:F than the root may be labeled by formulas as well). Most of the rules are given in the notation "$t_1{:}F_1,\ t_2{:}F_2 \ldots t_n{:}F_n \Rightarrow t{:}F$", meaning that if $t_1{:}F_1,\ t_2{:}F_2 \ . \ . \ . \ t_n{:}F_n$ are constructions then so is t:F.

As a parameter of the definition given below, we assume that a collection of *proof procedures* $\gamma_1, \gamma_2 \ldots \gamma_n$ has been given for lemmas $F_1, F_2, \ldots F_n$. In the current context - that is to say, in the context of a discussion of constructions - a proof procedure γ for a formula $\forall x_1, x_2 \ldots x_k F(x_1, x_2 \ldots x_k)$ is a procedure which, when given terms $t_1, t_2 \ldots t_k$ of L, either returns "FAIL", or else supplies a construction $t{:}F(t_1,t_2 \ldots t_k)$, where the construction t does not itself make use of any lemmas. Further, we require that $\gamma(t_1,t_2 \ldots t_k)$ be a closed construction whenever $t_1,t_2 \ldots t_k$ are closed. Thus a proof procedure in the context of constructions plays the same role as the proof procedures for natural deduction proofs discussed in section 2.5. We assume also that names $f_1,f_2, \ldots f_n$ of appropriate arities from the set D of defined symbols (see page 34) have been assigned as labels of the proof procedures $\gamma_1,\gamma_2 \ldots \gamma_n$.

The clauses of the inductive definition of the notion of a construction are as follows. Note that we have required that the axioms which appear in constructions be Harrop formulas (clause 2 below). Henceforward, we will also assume that the proofs which we consider contain only Harrop axioms, since proofs which do not satisfy this requirement are not in any case of much computational interest.

(1) $\alpha{:}A$ is a construction for any proof variable α and any formula A [assumption].

(2) If F is any Harrop formula, then $\#{:}F$ is a construction. [axiom]

(3) If f labels a proof procedure for the lemma $A = \forall x_1,x_2 \ldots x_k\varphi$, then $f{:}A$ is a construction. [lemma]

(4) $t_1{:}A,\ t_2{:}B \Rightarrow \langle t1,t2\rangle{:}A\wedge B$ [$\wedge$-introduction]

(5) (a) $t{:}A\wedge B \Rightarrow \pi_1(t){:}A$ (b) $t{:}A\wedge B \Rightarrow \pi_2(t){:}B$ [$\wedge$-elimination]

(6) (a) $t{:}A \Rightarrow OI_1(t){:}A\vee B$ (b) $t{:}B \Rightarrow OI_2(t){:}A\vee B$ [$\vee$-introduction]

(7) Let $t_1{:}A\vee B$, $t_2{:}C$, $t_3{:}C$ be constructions, and let α be a proof variable. Suppose that free occurences of α in t_2 are assigned the formula A, and that free occurences of α in t_3 are assigned the formula B. Then $OE(\alpha,t_1,t_2,t_3){:}C$ is a construction. [$\vee$-elimination]

(8) Let $t{:}B$ be a construction in which free occurences of the proof variable α are assigned the formula A. Then $(\lambda\alpha.t){:}A\supset B$ is a construction. [$\supset$-introduction]

(9) $t_1{:}A\supset B,\ t_2{:}A \Rightarrow t_1(t_2){:}B$ [$\supset$-elimination]

(10) Let $t{:}A$ be a construction with the property that no variable of the vector of variables $\underline{x}$ appears free in any of the formulas assigned to the free proof variables of t. Then $(\lambda\underline{x}.t){:}\forall\underline{x}A$ is a construction. [$\forall$-introduction]

(11) $t_1{:}\forall\underline{x}A(x) \Rightarrow t_1(t_2){:}A[\underline{x}\leftarrow\underline{t}_2]$ where t_2 is any vector of terms of L [$\forall$-elimination]

(12) $t_2{:}A[x \leftarrow t_1] \Rightarrow EI(t_1,t_2){:}\exists xA$ [$\exists$-introduction]

(13) Let $t_1{:}\exists xA$, and $t_2{:}C$ be constructions satisfying the following restrictions. (a) Free occurences of the proof variable α in t_2 are assigned the formula A. (b) Let F be any formula which is assigned to a free proof variable of t_2 other than α. Then x may not appear free in F. (c) The variable x may not appear free in C. Then $EE(x,\alpha,t_1,t_2){:}C$ is a construction. [$\exists$-elimination]

Note that, since we do not distinguish between formulas which differ only in the names of their bound variables, the identity of the variables bound by λ and the variables bound by $\forall$ in "$(\lambda \underline{x}.t){:}\forall \underline{x}A$" of rule (9) is a matter notational convenience and not a requirement. That is to say, for any new tuple of variables $\underline{y}$, "$\lambda \underline{y}.(t[\underline{x} \leftarrow \underline{y}]){:}\forall \underline{x}A$" and "$(\lambda \underline{x}.t){:}\forall \underline{x}A$" are equivalent labeled p-terms and have equal standing as well formed constructions. A similar remark applies to the construction of rule (11).

Arbitrary natural deduction proofs can be r written as constructions by a straight-forward extension of the methods which apply to proofs of pure implicational logic. Specifically, one starts out with an assignment of proof variables α_A to formulas A. Then the map Γ from proofs to constructions is defined by induction on the structure of proofs just as it was in section 3.1. What Γ does is (1) replace each assumption [A] by the variable α_A assigned to A, (2) replace axioms by the special constant #, (3) replace lemmas by the defined symbols which label their proof procedures, (3) replace each inference rule by the corresponding constructor, and finally (4) label each node of the p-term by the formula which occurs at the corresponding node of the proof tree. The passage in the other direction is even more straight-pforward: to go from a construction to a natural deduction proof one keeps the formulas and constructors which label the tree, but the proof variables are thrown away. The clauses of the inductive definition of $\Gamma$are given below, using the notation $\Gamma{:}\ \Pi \Rightarrow t{:}F$ to indicate that the value of Γ applied to Π is the construction t:F.

(1) Base case: $\Gamma{:}\ [A] \Rightarrow \alpha_A{:}A$

(That is, Γ when applied to a proof which consists simply of an assumption [A] yields the construction $\alpha_A{:}A$.)

(2) Base case: $\Gamma{:}\ A \Rightarrow \#{:}A$, where A is an axiom.

(Γ when applied to a proof which consists of an axiom A yields the construction #:A.)

(3) Base case: Γ: A $\Rightarrow$ f, where A is a lemma, and where f labels a proof procedure for the formula F. (Evidently, it is possible - by virtue of the effective character of Γ itself - to convert any natural deduction style proof procedure into a construction style proof procedure.)

(4)

$$\Gamma: \quad \wedge I\frac{\begin{matrix}\Pi_1 & \Pi_2\\ A & B\end{matrix}}{A\wedge B} \quad\Rightarrow\quad \langle\Gamma(\Pi_1),\Gamma(\Pi_2)\rangle : A\wedge B$$

(5a)

$$\Gamma: \quad \wedge E\frac{\begin{matrix}\Pi\\ A\wedge B\end{matrix}}{A} \quad\Rightarrow\quad \pi_1(\Gamma(\Pi)):A$$

(5b)

$$\Gamma: \quad \wedge E\frac{\begin{matrix}\Pi\\ A\wedge B\end{matrix}}{B} \quad\Rightarrow\quad \pi_2(\Gamma(\Pi)):B$$

(6a)

$$\Gamma: \quad \vee I\frac{\begin{matrix}\Pi\\ A\end{matrix}}{A\vee B} \quad\Rightarrow\quad OI_1(\Gamma(\Pi)):A\vee B$$

(6b)

$$\Gamma: \quad \vee I\frac{\begin{matrix}\Pi\\ B\end{matrix}}{A\vee B} \quad\Rightarrow\quad OI_2(\Gamma(\Pi)):A\vee B$$

(7)

$$\Gamma: \quad \vee E\frac{\begin{matrix}\Pi_1 & \Pi_2 & \Pi_3\\ A\vee B & C & C\end{matrix}}{C} \quad\Rightarrow\quad OE(\beta,\ \Gamma(\Pi_1),\ \Gamma(\Pi_2)[\alpha_A\leftarrow\beta],\ \Gamma(\Pi_3)[\alpha_B\leftarrow\beta]):C$$

where the "new" variable β does not occur free in $\Gamma(\Pi_2)$ or in $\Gamma(\Pi_3)$.

(8)

$$\Gamma: \quad \supset\! I\,\frac{\begin{matrix}[A]\\ \Pi \\ B\end{matrix}}{A \supset B} \qquad \Rightarrow \qquad \lambda v_A.\ \Gamma(\Pi):A \supset B$$

(9)

$$\Gamma: \quad \supset\! E\,\frac{\begin{matrix}\Pi_1 \\ A\end{matrix} \qquad \begin{matrix}\Pi_2 \\ A \supset B\end{matrix}}{B} \qquad \Rightarrow \qquad (\Gamma(\Pi_1))(\Gamma(\Pi_2)):B$$

(10)

$$\Gamma: \quad \forall I\,\frac{\begin{matrix}\Pi \\ A\end{matrix}}{\forall \underline{x} A} \qquad \Rightarrow \qquad \lambda \underline{x}.\Gamma(\Pi): \forall \underline{x} A$$

(11)

$$\Gamma: \quad \forall E\,\frac{\begin{matrix}\Pi \\ \forall \underline{x} A\end{matrix}}{A[\underline{x} \leftarrow \underline{t}]} \qquad \Rightarrow \qquad (\Gamma(\Pi))(\underline{t}):A[\underline{x} \leftarrow \underline{t}]$$

(12)

$$\Gamma: \quad \exists I\,\frac{\begin{matrix}\Pi \\ A[x \leftarrow t]\end{matrix}}{\exists x A} \qquad \Rightarrow \qquad EI(t,\Gamma(\Pi)): \exists x A$$

(13)

$$\Gamma: \quad \exists E\,\frac{\begin{matrix}\Pi_1 \\ \exists x A\end{matrix} \quad \begin{matrix}\Pi_2 \\ C\end{matrix}}{C} \qquad \Rightarrow \qquad EE(x,\alpha_A,\Gamma(\Pi_1),\Gamma(\Pi_2)):C$$

The construction notation U_c for the upper bound proof U of section 2.8 is given below as an example.

$$\begin{array}{l}
\mathrm{OE}(\alpha,\mathrm{LESSD}(x,1),\\
\quad \mathrm{EI}(y+1,\#(\alpha))\\
\qquad \mathrm{OE}(\beta,\mathrm{LESSD}(y,1),\\
\qquad\quad \mathrm{EI}(x+1,\#(\beta)),\\
\qquad\quad \mathrm{EI}(2xy,\#(\langle\alpha,\beta\rangle))))):\exists z\Psi(x,y,z)
\end{array}$$

In the above presentation of U_c, only the root node is explicitly labeled by a formula; we have neglected to specify the formulas which are attached to the various subterms of the construction. A complete description of the construction is as follows, where the formula F which labels each subterm t is specified using the notation "t:F".

$$\begin{array}{l}
\mathrm{OE}(\alpha,\{\mathrm{LESSD}:\forall xy(x\leq y\vee y<x)\}(x,1):x\leq 1\vee x>1,\\
\quad \mathrm{EI}(y+1,\{\#:x\leq 1\supset\Psi(x,y,y+1)\}(\beta:x\leq 1):\Psi(x,y,y+1)):\exists z\Psi(x,y,z)\\
\quad \mathrm{OE}(\beta,\{\mathrm{LESSD}:\forall xy(x\leq y\vee y<x)\}(y,1):y\leq 1\vee y>1,\\
\qquad \mathrm{EI}(x+1,\{\#:y\leq 1\supset\Psi(x,y,x+1)\}(\beta:y\leq 1):\Psi(x,y,x+1)):\exists z\Psi(x,y,z),\\
\qquad \mathrm{EI}(2xy,(\#:(x>1)\wedge(y>1)\supset\Psi(x,y,2xy))\\
\qquad\qquad\qquad (\langle\alpha:\{x>1\},\beta:\{y>1\}\rangle:\{x>1\wedge y>1\}):\Psi(x,y,2xy))))):\exists z\Psi(x,y,z)
\end{array}$$

3.3 *Substitution*

The effect of the principle of "substitution of equals for equals" can be obtained by the use of a scheme of Harrop axioms (as was done for FALSE - elimination; see section 2.1). However, it is more convenient for our purposes to include the following inference rule which expresses this principle directly.

Substitution:

$$\frac{t_1=t_2 \qquad A}{A[t_1\rightsquigarrow t_2]} \qquad\text{and}\qquad \frac{t_1=t_2 \qquad A}{A[t_2\rightsquigarrow t_1]}$$

On the p-calculus side, a new constructor: $\mathrm{SB}(t_1,t_2)$ is added, and the clauses

$$\Gamma:\quad \mathrm{SB}\frac{\begin{matrix}\Pi_1 & \Pi_2\\ t_1=t_2 & A\end{matrix}}{A[t_1\rightsquigarrow t_2]} \quad\Rightarrow\quad \mathrm{SB}(\Gamma(\Pi_1),\Gamma(\Pi_2)):A[t_1\rightsquigarrow t_2]$$

$$\Gamma:\quad \mathrm{SB}\dfrac{\begin{array}{cc}\Pi_1 & \Pi_2\\ t_1=t_2 & A\end{array}}{A[t_2 \leadsto t_1]} \qquad\Rightarrow\qquad \mathrm{SB}(\Gamma(\Pi_1),\Gamma(\Pi_2)):A[t_2 \leadsto t_1]$$

are added to the definition of the map Γ from natural deduction proofs to constructions.

3.4 Recursive constructions

Recursive definitions of functions are commonly used for describing computational methods, both in mathematics, and in automatic computation. Most programming languages allow defintion by recursion, and in purely applicative languages, such as pure LISP [McCarthy et al, 1962], the principal constructors used in building up programs are just function application, and recursive definition.

We too will make use of definition by recursion. Specifically, we will allow (mutually) recursive definitions of the form:

$$\begin{array}{l} f_1 \leftarrow t_1{:}A_1 \\ f_2 \leftarrow t_2{:}A_2 \\ \\ \;\;\ldots \\ f_n \leftarrow t_n{:}A_n \end{array}$$

where the $\{f_i\}$ are defined symbols, and the $\{t_i\}$ are constructions in which $f_1 \ldots f_n$ may appear, and the $\{A_i\}$ are universal formulas. The following restrictions apply: (1) each construction $t_i{:}A_i$ must be closed, and (2) each occurence of a defined name f_i in any t_j must have A_i as its attached formula.

Putting the matter more formally, we implement definitions by recursion in the following way. A parameter of the definition of the class of constructions is the set of assignments made to defined symbols. Until now, those assignments have been proof procedures with appropriate charactersistics. Henceforth, we will allow constructions as well as proof procedures to be assigned as values of defined symbols, subject to the restrictions described in the last paragraph. Of course, each defined symbol may be assigned only one value, whether a proof procedure or a construction. We will refer to a set of assignments of constructions and proof procedures to defined symbols as a "system of definitions" or a "system of lemmas". The system of definitions which is in effect for the purposes of any particular discussion will be referred to as the "current system of definitions".

If one switches back from the terminology of constructions to that of natural deduction proofs, then the "recursive proofs" which correspond to recursive constructions are proofs which use their own end-formulas as lemmas. An example of a computationally useful recursive proof is as follows.

Let pred denote the predecessor function on natural numbers (it does not matter what value is chosen for pred(0)). Then one formulation of the induction principle for the formula $\varphi(x)$ is as follows:

$$\mathrm{IND}_{\varphi}\colon\ \forall x(\ \{\varphi(0)\wedge\forall y\,(y\neq 0\wedge\varphi(\mathrm{pred}\ y))\supset\varphi(y))\}\supset\varphi(x))$$

The following is a a recursive proof of IND - a proof in which IND itself is used as a lemma. We will need an abbreviation. Let H be the formula:

$$\forall y\,(y\neq 0\wedge\varphi(\mathrm{pred}\ y))\supset\varphi(y))$$

Then IND_{φ} is just $\forall x(\varphi(0)\ \wedge\ H\ \supset\ \varphi(x))$. The proof, then, is as follows.

$$\forall\mathrm{I}\dfrac{\supset\mathrm{I}\dfrac{\vee\mathrm{E}\dfrac{\forall\mathrm{E}\dfrac{\forall xy(x=y\vee x\neq y)}{x=0\vee x\neq 0}\quad \mathrm{SB}\dfrac{[x=0]\quad \wedge\mathrm{E}\dfrac{[\varphi(0)\wedge H]}{\varphi(0)}}{\varphi(x)}\quad \supset\mathrm{E}\dfrac{\wedge\mathrm{I}\dfrac{[x\neq 0]\quad \supset\mathrm{E}\dfrac{[\varphi(0)\wedge H]\quad \forall\mathrm{E}\dfrac{\mathrm{IND}\varphi\colon\forall x(\ \varphi(0)\wedge H\supset\varphi(x))}{\varphi(0)\wedge H\supset\varphi(\mathrm{pred}\ x)}}{\varphi(\mathrm{pred}\ x)}}{x\neq 0\wedge\varphi(\mathrm{pred}\ x)}\quad \forall\mathrm{E}\dfrac{\wedge\mathrm{E}\dfrac{[\varphi(0)\ \wedge\ H]}{H}}{x\neq 0\wedge\varphi(\mathrm{pred}\ x)\ \supset\varphi(x)}}{\varphi(x)}}{\varphi(x)}}{\varphi(0)\wedge H\ \supset\ \varphi(x)}}{\forall x(\varphi(0)\ \wedge\ H\ \supset\ \varphi(x))}$$

In the notation of constructions, the above proof of IND_{φ} looks like this:

$$\begin{aligned}\mathrm{IND}_{\varphi}\ =\ & \lambda x.\lambda\alpha.\mathrm{OE}(\beta,\mathrm{EQD}(x,0),\\ & \quad \mathrm{SB}(\beta,\pi_1(\alpha)),\\ & \quad \{(\pi_2(\alpha))(\mathrm{pred}\ \ x)\}(\langle\beta,(\mathrm{IND}_{\varphi}(\mathrm{pred}\ \ x))(\alpha)\rangle))\end{aligned}$$

where EQD is a proof procedure for the formula $\forall xy(x=y\vee x\neq y)$; EQD returns the construction, "$OI_1(\#:t_1=t_2):t_1=t_2 \vee t_1\neq t_2$" if t_1 and t_2 are closed terms with $t_1=t_2$ and "$OI_2(\#:t_1\neq t_2):t_1=t_2 \vee t_1\neq t_2$" if t_1 and t_2 are closed terms with $t_1\neq t_2$.

The usefulness of this construction derives from the fact that the value to which the lemma IND is applied is the predecessor of the value to which the theorem IND is applied. The construction may be executed when applied to a particular numeral in the same way that a recursively defined function is run: by repeated replacements of the defined name IND by its definition. As a consequence of the fact that the value passed to succesive recursive calls to IND is constantly decreasing, this mode of execution will terminate (under the right reduction order), yielding a construction in which no reference to IND any longer appears. The details of this process will be discussed later (section 3.5).

A somewhat simpler way of achieving the effect of induction by the use of recursive proofs is as follows. Suppose that one has a proof Π_1 of $\varphi(0)$, and a proof Π_2 of $\forall y\ (y\neq 0 \wedge \varphi(\text{pred } y)) \supset \varphi(y))$. Then the following recursive proof P_φ of $\forall x\varphi(x)$ is adequate to the same computational purposes as is the above proof of IND_φ.

$$\forall I\dfrac{\vee E\dfrac{\forall E\dfrac{\forall xy(x=y\vee x\neq y)}{x=0 \vee x\neq 0}\qquad SB\dfrac{[x=0]\quad \overset{\Pi_1}{\varphi(0)}}{\varphi(x)}\qquad \supset E\dfrac{\wedge I\dfrac{[x\neq 0]\quad \forall E\dfrac{P_\varphi:\forall x(\ \varphi(x))}{\varphi(\text{pred } x)}}{x\neq 0 \wedge \varphi(\text{pred } x)}\qquad \forall E\dfrac{\overset{\Pi_2}{\forall y(y\neq 0 \wedge \varphi(\text{pred } y) \supset \varphi(y))}}{x\neq 0 \wedge \varphi(\text{pred } x) \supset \varphi(x)}}{\varphi(x)}}{\varphi(x)}}{\forall x\varphi(x)}$$

The construction notation for P_φ is as follows, where t_1 is the construction notation for Π_1, and t_2 the construction notation for Π_2.

$$\begin{aligned} P_\varphi \ = \ & \lambda x.OE(\alpha,EQD(x,0),\\ & \qquad SB(\alpha,t_1),\\ & \qquad (t_2(x))(\langle\alpha,P_\varphi(\text{pred } x)\rangle)) \end{aligned}$$

Suppose that a system S of lemmas has the property that every axiom in sight is true in a particular model M. That is to say, we suppose that all the axioms which appear in constructions of S, and all axioms which appear in the constructions generated by proof procedures of S, are true in M. Note that these conditions are still not sufficient to guarantee that constructions built up from lemmas of S will have end-formulas which are true in M. The reason for this is that constructions can take the form of circular arguments. Consider, for example, recursively defined construction : "$f{:}\forall xA \leftarrow f{:}\forall xA$", which of course provides no evidence at all for the truth of $\forall xA$. In order to verify the truth of the end-formula of a construction, or the correctness of the computation described by the construction, it is not sufficient to verify the axioms which are used (directly or indirectly) by the construction. It is also necessary to verify the truth of the lemmas which are used, even though (recursive) constructions for those lemmas have been supplied.

3.5 Operations on constructions

In this section, the various elementary operations which are involved in the computational use of constructions are described. These operations are: (a) the normalization reductions, and (b) the pruning operations. These operations are arrived at by direct translation into construction notation of the operations on natural deduction proofs given in sections 2.4, 2.7.

$\wedge$-reduction:

$$\pi_1(\langle t_1{:}A, t_2{:}B\rangle{:}A\wedge B){:}A \quad\Rightarrow\quad t_1{:}A$$

$$\pi_2(\langle t_1{:}A, t_2{:}B\rangle{:}A\wedge B){:}B \quad\Rightarrow\quad t_2{:}B$$

$\vee$-reduction:

$$OE(\alpha, OI_1(t_1{:}A){:}A\vee B, t_2{:}C, t_3{:}C){:}C \quad\Rightarrow\quad t_2[\alpha\leftarrow t_1]{:}C$$

$$OE(\alpha, OI_2(t_1{:}B){:}A\vee B, t_2{:}C, t_3{:}C){:}C \quad\Rightarrow\quad t_3[\alpha\leftarrow t_1]{:}C$$

$\supset$-reduction:

$$\{(\lambda\alpha.(t_1{:}B)){:}A\supset B\}(t_2{:}A)\ {:}B \quad\Rightarrow\quad t_1[\alpha\leftarrow t_2]{:}B$$

$\forall$-reduction:

$$\{(\lambda\underline{x}.(t_1{:}A)){:}\forall\underline{x}A\}(\underline{t}_2)\ {:}A \quad\Rightarrow\quad t_1[\underline{x}\leftarrow\underline{t}_2]{:}A[\underline{x}\leftarrow\underline{t}_2]$$

$\exists$-reduction:

$$EE(x, \alpha, EI(t_1, t_2{:}A){:}\exists xA), t_3{:}C) \quad\Rightarrow\quad (t_3[x\leftarrow t_1])[\alpha\leftarrow t_2]{:}C$$

Lemma-reduction:

$\{(f{:}\forall\underline{x}A)(\underline{t})\}{:}A[x\leftarrow\underline{t}] \quad\Rightarrow\quad \gamma(\underline{t}){:}A[\underline{x}\leftarrow\underline{t}]$

condition: f has been assigned the proof procedure γ, and $\gamma(\underline{t})\neq$FAIL.

$\{(f{:}\forall\underline{x}A)(\underline{t})\}{:}A[x\leftarrow\underline{t}] \quad\Rightarrow\quad t'(\underline{t}){:}A[\underline{x}\leftarrow\underline{t}]$

condition: $\underline{t}$ is closed, and f has been assigned the construction t'.

In addition, a reduction rule for the new substitution inference of section 3.3 is needed:

$SB(t_1{:}(t_3=t_4),t_2{:}A[x\leftarrow t_3]){:}A[x\leftarrow t_4] \quad\Rightarrow\quad t_2{:}A[x\leftarrow t_4]$

condition: t_1 may not contain free proof variables.

The effect of SB-reduction is to take the construction t_2 and simply replace the formula $A[x\leftarrow t_3]$ attached to its root by $A[x\leftarrow t_4]$, and thus dispensing with the SB inference rule. Evidently, if t_3 and t_4 are distinct terms, then the result of applying SB-reduction to a construction will be a labeled p-term which is no longer a construction. However, if the axioms which appear in t_1 are correct (in some particular model) then t_3 and t_4 will denote the same object in the model, and in this sense the formulas A and A have the same meaning. In fact, nothing will go wrong if we fail to distinguish between formulas which differ only by substitution of one term of L by another which denotes the same object. More formally, relative to any particular model, we may expand the class of constructions to include all of those labeled p-terms which can be arrived at by substitution of "equals for equals" in formulas. This mars the uniformity of our treatment in that introduces model-theoretic considerations into the defintion of the notion of a construction, whereas that notion has been purely syntactic until now. However, as we have said, none of our results are affected.

The pruning reductions are as follows:

$OE(\alpha,t_1{:}A\vee B,t_2{:}C,t_3{:}C)\ {:}C \quad\Rightarrow\quad t_2{:}C$

Condition: α does not appear free in t2.

$OE(\alpha,t_1{:}A\vee B,t_2{:}C,t_3{:}C)\ {:}C \quad\Rightarrow\quad t_3{:}C$

Condition: α does not appear free in t_3.

Note that each of the operations described in this section applies to the untyped part of a construction independently of the attached formulas, in the following sense. Let t:F be a construction, and let t':F be the result of applying one of the operations listed above to t:F. Further, let untyp(r) for any construction r denote the untyped p-term which forms the "skeleton" of r - that is to say, untyp(r) is arrived at from r by removing the formulas which label the nodes of r. Then untyp(t') can be computed from untyp(t) alone. As was mentioned earlier in the context of the typed λ-calculus, the consequence of this observation for computational purposes is that the execution and pruning of a construction may be carried out by treating only its untyped part; the attached formulas need not be carried around in the course of the computation. In order to clarify the manner in which the various operations apply to untyped p-terms, we list those operations below with the type information left out.

$\wedge$-reduction:

$\pi_1(\langle t_1,t_2\rangle)$	$\Rightarrow$	t_1
$\pi_2(\langle t_1,t_2\rangle)$	$\Rightarrow$	t_2

$\vee$-reduction:

$OE(\alpha,OI_1(t_1),t_2,t_3)$	$\Rightarrow$	$t_2[\alpha\leftarrow t_1]$
$OE(\alpha,OI_2(t_1),t_2,t_3)$	$\Rightarrow$	$t_3[\alpha\leftarrow t_1]$

$\supset$-reduction:

$(\lambda\alpha.t_1)(t_2)$	$\Rightarrow$	$t_1[\alpha\leftarrow t_2]$

$\forall$-reduction:

$(\lambda\underline{x}.t_1)(\underline{t}_2)$	$\Rightarrow$	$t_1[\underline{x}\leftarrow\underline{t}_2]$

$\exists$-reduction:

$EE(x,\alpha,EI(t_1,t_2),t_3)$	$\Rightarrow$	$(t_3[x\leftarrow t_1])[\alpha\leftarrow t_2]$

Lemma-reduction:

$f(\underline{t})$	$\Rightarrow$	$\gamma(\underline{t})$

condition: $\underline{t}$ is closed, and f has been assigned the proof procedure γ.

$f(\underline{t}) \Rightarrow t'(\underline{t})$

condition: $\underline{t}$ is closed, and f has been assigned the construction t'.

SB-reduction:

$SB(t_1,t_2) \Rightarrow t_2$

condition: t_1 does not contain free proof variables.

Pruning:

$OE(\alpha,t_1,t_2,t_3) \Rightarrow t_2$

Condition: α does not appear free in t_2.

$OE(\alpha,t_1,t_2,t_3) \Rightarrow t_3$

Condition: α does not appear free in t_3.

3.6 Results about constructions

As was emphasized in section 2.9 in the context of natural deduction proofs, the results and conditions which are relevant to the computational use of proof normalization are of several independent kinds. Specifically, there are results concerning

(1) the syntactic soundness of the normalization reductions,

(2) the semantic soundness of proofs,

(3) special properties of proofs in normal form, and finally,

(4) the termination of reduction sequences.

The conditions upon which the results in one catagory depend, and the proofs of those results, are for the most part unrelated to the conditions and proofs which come up in the other catagories. In sections 2.3 and 2.9, results about normalization were stated without proof as they apply to natural deduction proofs. In this section, we will prove or sketch proofs of the corresponding results which apply to constructions.

Let R be the set of all the elementary operations on constructions which were described in the last section. We have:

Proposition 1: Each of the operations of R yields a construction when applied to a construction.

Proposition 2: Each of the operations of R yields a closed construction when applied to a closed construction.

Proposition 3: Each of the operations R preserves the end-formula of the construction to which it is applied.

The above propositions can be verified by inspection.

Definition: A construction t:F is a *valid construction* relative to a model M if the universal closure of each axiom and each lemma which appears in t is *true* in M

Definition: A system of lemmas is *valid relative to M* if each construction which appears in the system is valid relative to M, and if each proof procedure γ which appears in the system returns only valid constructions.

Proposition 4 (Soundness): Suppose that t:F is a construction with free proof variables $\alpha_1 \ldots \alpha_n$, and free object variables $x_1 \ldots x_n$. Suppose further that t:F is valid relative to M. Let $A_1 \ldots A_n$ be the formulas which are attached to the free proof variables $\alpha_1 \ldots \alpha_n$. Then $\forall x_1\ x_2 \ldots x_n((A_1 \wedge A_2 \ldots A_n) \supset F)$ is *true* in M.

Proof: Induction on the structure of constructions; each of the rules by which constructions are built up preserves soundness.

Proposition 5: Suppose that (a) t:F is valid relative to M, and (b) the current system of lemmas is valid relative to M. Then the result of applying any of the operations of section 3.5 to t:F is a valid construction.

Proof: Observe that, with the exception of the lemma-reduction operation, all operations in R modify the axioms appearing in constructions only by instantiating free variables which appear in those axioms. The proposition follows.

Definition: We classify constructors as either "introduction constructors" or "elimination constructors" according to their correspondence to the introduction rules and elimination rules of natural deduction. The introduction constructors are PAIR, OI_1, OI_2, λ-ABSTRACTION, and EI, and the elimination constructors are $\pi 1$, $\pi 2$, OE, APPLY, EE, and SB. (SB "eliminates" an equation.)

Theorem 3.1: Let t:F be a closed construction in normal form where F is not a Harrop formula. Then either t:F is a lemma (ie has the form f:F where f is a defined symbol), or the main constructor of t is an introduction constructor.

Proof: By induction on the structure of proofs. Suppose that t:F is a normal closed construction where F is not Harrop. Base case: If t is an "atomic" construction consisting of one node, then it must be either (1) an assumption, (2) a Harrop axiom, (3) a lemma f:F. Cases (1) and (2) are impossible: (1) because t is closed, (2) because is F not a Harrop formula. Thus case (3) must hold, and so the base of the induction is verified. Furthermore we have verified that any t:F which is not a lemma (and which satisfies the hypotheses of the theorem) must have a "main" constructor, whether it be an introduction contructor or an elimination constructor, since t cannot be atomic. For the induction step, we assume that the proposition holds for each sub-construction of t:F, and then derive a contradiction from the supposition that the main constructor of t is an elimination constructor. There are 6 cases to consider. Suppose that the main constructor of t is (1) π_1, (2) π_2, (3) OE, (4) APPLY, (5) EE, (6) SB. Then t:F has one of the forms (1) $\pi_1(t_1{:}F\wedge G){:}F$ (2) $\pi_2(t_1{:}G\wedge F){:}F$ (3) $OE(\alpha,t_1{:}A\vee B,t_2{:}F,t_3{:}F){:}F$ (4) $(t_1{:}G\supset F)(t_2{:}G){:}F$ or $(t_1{:}\forall xA)(t_2){:}A[x\leftarrow t_2]$ (5) $EE(t_1{:}\exists xA,t_2{:}F){:}F$ (6) $SB(t_1,t_2){:}F$. In case (6) t_1 is closed, and therefore SB reduction can be applied, contrary to the hypothesis that t is in normal form. In all other cases, t_1 is closed and has a non-Harrop end-formula, so the induction hypothesis applies. Thus t_1 is either a lemma, or else has an introduction constructor as its main constructor. If t_1 is a lemma, then t must have the form $t_1(\underline{t_2})$, where $\underline{t}_2$ is closed, and thus lemma-reduction could have been applied, contrary to the hypothesis that t is normal. If t_1 has an introduction constructor as its main constructor, then, by virtue of the form of the end-formula of t_1, that main constructor must be (1) PAIR, (2) PAIR, (3) OI_1 or OI_2, (4) λ-ABSTRACTION, (5) EI. But then one of the reduction rules (1) $\wedge$-reduction, (2) $\wedge$-reduction, (3) $\vee$-reduction, (4) $\supset$-reduction or $\forall$-reduction, (5) $\exists$-reduction, can be applied to t, again contrary to the hypothesis that t is in normal form.

Corollary 1: If $t{:}\exists xA$ is closed and normal, then t has the form $EI(t_1,t_2{:}A[x\leftarrow t_1]){:}\exists xA$. If, in addition, t is valid relative to M, then $A(t_1)$ holds in M.

Corollary 2: If $t{:}A\vee B$ is closed, and normal, then t has one of the forms $OI_1(t_1{:}A){:}A\vee B$, or $OI_2(t_1{:}B){:}A\vee B$. If, in addition, t is valid relative to M, then, in the first case, A holds in M, and in the second case B holds in M.

The following corollary of the various results given above establishes the usefulness of normalization for computational purposes, and the conditions for the partial correctness of a construction regarded as a computational description.

Corollary 3: Let $t_1{:}\forall x\exists y\varphi(x,y)$ be a closed construction and let t_2 be a closed term of L. Suppose that some sequence of applications of operations of R to the construction

$t_1(t_2):\exists y\varphi(t_2,y)$ yields a normal construction t'. Then t' has the form $EI(t_3,t_4):\exists y\varphi(t_2,y)$. Further, if t_1 is valid relative to a model M, and the current system of lemmas is valid relative to M, then $\varphi(t_2,t_3)$ is true in M.

Corollary 3 shows that normalization constitutes a satisfactory means for executing a construction $t_1:\forall x\exists y\varphi(x,y)$ in the sense that if one "puts in" a value t_2 for x, and if normalization terminates, then a value t_3 for y comes out. In addition, the corollary shows that t_1 regarded as a program is partially correct with respect to the input-output specification φ, under the condition that all lemmas and axioms in sight are true. Thus the verification of the partial correctness of an algorithm expressed by a construction is a matter of establishing the truth of formulas which appear explicitly in the construction and in its system of lemmas. As a consequence, the passage from a construction to its "verfication conditions" is simpler for constructions than for computational descriptions of a more conventional kind.

We turn now to the question of termination.

Definition: A construction t:F has the "termination property" if every sequence of applications of operations in R to t is finite. That is, there is no infinite sequence of terms $t_1, t_2, \ldots$ such that $t_1=t$, and such that t_{i+1} arises from t_i by the application of one of the reductions of R.

Theorem 3.2 (Termination): Suppose that t:F is a recursion-free construction in in the sense that all defined symbols which appear in t are assigned proof procedures and not constructions. Then t has the termination property.

The standard proof of the termination of normalization for the predicate calculus (see eg Prawitz[1969]) or equivalently for the typed λ-calculus (see Troelstra [1973A]) applies to the calculus of constructions with only minor technical modifications. Therefore, we omit the proof of theorem 3.2 here.

Evidently, if recursively defined symbols appear in a construction, the termination theorem no longer applies. Indeed, there are recursive definitions of a symbol f (such as the looping definition "$f:\forall xA \leftarrow f:\forall xA$") which have the property that *no* finite sequence of reductions of f(t) where t is closed can lead to a normal form.

Consider the formulation of first order arithmetic which is arrived at by taking the members of the schema IND_φ as the *only* recursive constructions. Even here the termination property fails. The reason for this is that one is free to repeatedly apply lemma-reduction to $IND\varphi(t)$, with t closed, without performing any other reductions, and this process will not terminate. However, termination can be guaranteed if an additional restriction is made on lemma-reduction as it applies to the induction schema. Namely, we require that if lemma-reduction is applied to $IND_{\varphi}(t)$ yielding

$$t' = \lambda x.\lambda\alpha.OE(\beta,EQD(x,0), \\ SB(\beta,\pi_1(\alpha)), \\ \{(\pi_2(\alpha))(\text{pred } x)\}(\langle\beta,(IND_{\varphi}(\text{pred } x))(\alpha)\rangle))\ (t)$$

then t' must be brought immediately into one of the two forms

$$\lambda\alpha.(\pi_1(\alpha))$$

or

$$\lambda\alpha.\ (\{(\pi_2(\alpha))(\text{pred } t)\}(\langle\#,(IND_{\varphi}(\text{pred } t))(\alpha)\rangle))$$

(depending on whether the value of t is zero) before any other reductions are applied. (This immediate reduction of t' will involve one application of $\supset$-reduction, one application of lemma-reduction to EQD, one application of $\vee$-reduction, and perhpas one application of SB-reduction.) When this restriction is made, the effect of lemma-reduction together with the immediately succeeding reduction steps is very much like that of the induction-reduction rule in the usual formulation of normalization for first order arithemtic (see Prawitz[1965]). The restriction results in a system with the strong normalization property - a fact which can be *demonstrated by minor modification of the standard proof of strong normalization* for arithmetic (Troelstra[1973B]). Further, theorem 3.1 continues to apply, since we have restricted only the order in which reductions may be applied, and have not thereby modified the notion of a construction in normal form.

Leaving aside the special case of arithmetic, the situation is this. One may take any algorithm which is expressed by an (ordinary) recursive definition and reformulate it as a recursive construction; the form of the recursions in the construction will be identical to the form of the recursions in the original definition. (A concrete example is given in chapter 4.) If the ordinary recursive definition terminates under some particular order of evaluation (eg call-by-value or call-by-name), then so will the recursive construction under a corresponding reduction order for normalization. We do not propose to investigate here the general question of the termination of the normalization of recursive constructions. It is sufficient for the current purposes to observe that the particular reduction order which we use in the implementation (namely, the call-by-value order) terminates on the particular proof which concerns us (namely, the bin-packing proof of chapter 4).

3.7 Another reduction rule

An additional reduction rule beyond those so far mentioned is used in the normalization of the bin-packing proof of chapter 4 - namely, the permutation rule for the V-elimination inference:

$$\vee E\;\dfrac{\vee E\;\dfrac{\begin{matrix}\Pi_1\\ A\vee B\end{matrix}\quad\begin{matrix}[A]\\ \Pi_2\\ C\vee D\end{matrix}\quad\begin{matrix}[B]\\ \Pi_3\\ C\vee D\end{matrix}}{C\vee D}\qquad\begin{matrix}[C]\\ \Pi_4\\ E\end{matrix}\qquad\begin{matrix}[D]\\ \Pi_5\\ E\end{matrix}}{E}$$

$$\Rightarrow$$

$$\vee E\;\dfrac{\begin{matrix}\Pi_1\\ A\vee B\end{matrix}\qquad \vee E\;\dfrac{\begin{matrix}[A]\\ \Pi_2\\ C\vee D\end{matrix}\quad\begin{matrix}[C]\\ \Pi_4\\ E\end{matrix}\quad\begin{matrix}[D]\\ \Pi_5\\ E\end{matrix}}{E}\qquad \vee E\;\dfrac{\begin{matrix}[B]\\ \Pi_3\\ C\vee D\end{matrix}\quad\begin{matrix}[C]\\ \Pi_4\\ E\end{matrix}\quad\begin{matrix}[D]\\ \Pi_5\\ E\end{matrix}}{E}}{E}$$

In construction notation, this is:

$$OE(\alpha,OE(\beta,t_1:A\vee B,t_2:C\vee D,t_3:C\vee D),t_4:E,t_5:E):E \qquad \Rightarrow$$

$$OE(\beta,t_1:A\vee B,OE(\alpha,t_2:C\vee D,t_4:E,t_5:E):E,OE(\alpha,t_3:C\vee D,t_4:E,t_5:E):E):E$$

where it is assumed (without loss of generality) that β does not appear free in either t_4 or t_5.

None of the results concerning constructions given in section 3.6 is affected by the addition of this rule. This is immediate for all results concerning the properties of constructions in normal form, since any construction which is in normal form with respect to the reduction system which includes the new permutation rule is *a fortiori* in normal form with respect to the reduction system without this rule. The only result which needs checking is the termination theorem (theorem 3.2). But, as it happens, standard proofs of this theorem, such as that given in Prawitz[1969], treat reduction systems in which permutation reductions are included.

3.8 Effects of pruning on efficiency

To avoid misunderstanding: The principal evidence which we will provide concerning the *utility* of pruning in improving efficiency is the bin-packing example of chapter 4. But to help in choosing other examples where pruning is likely to be of use, it is desirable to illustrate the features on which the behaviour of pruning depends in a simple and abstract setting. With this in mind, we make the following formal points by means of schematic examples.

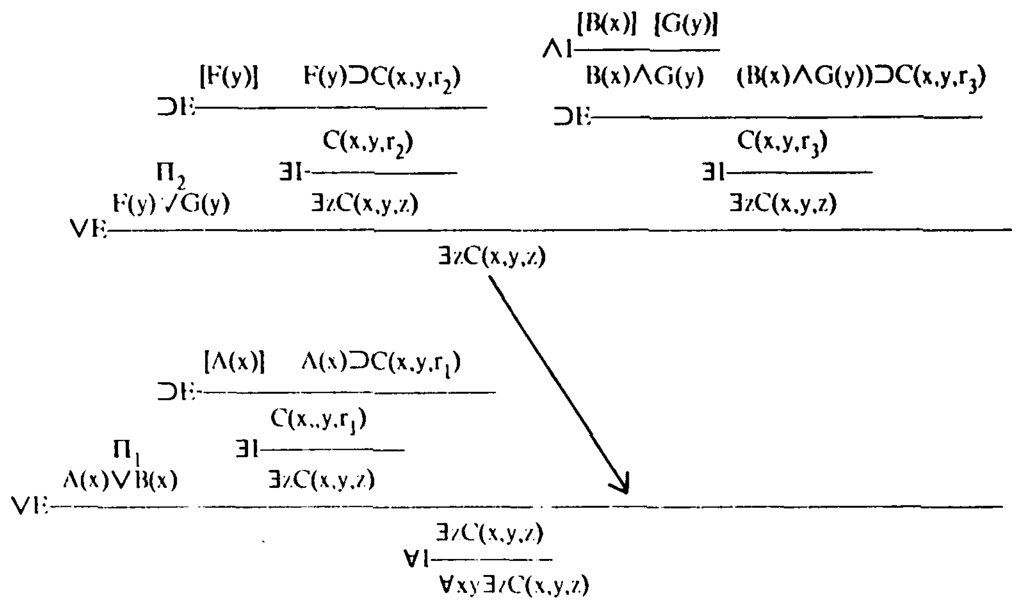
(1) Pruning can lead to a very large increase in the efficiency of an algorithm which has been specialized.

(2) Pruning can lead to a very large decrease in the efficiency of an algorithm.

(3) The inclusion of proofs of Harrop formulas can improve the effectiveness of pruning.

Consider, then, the following proof:

$P_1 =$



where the "results" r_1, r_2, r_3 are distinct terms of L. Note that the above schematic proof P_1 has the proof U of section 2.8 as an instance; take $A(x)$ = "$x \leq 1$", $B(x)$ = "$x > 1$", $F(y)$ = "$y \leq 1$", $G(y)$ = "$y > 1$", $C(x,y,z)$ = "$\Psi(x,y,z)$", r_1 = "$y+1$", r_2 = "$x+1$", r_3 = "$2xy$". P_1 when written as a construction is:

$$f = \lambda xy.\ OE(\alpha,t_1,EI(r_1,\#(\alpha)) \quad OE(\beta,t_2,EI(r_2,\#(\beta)), \quad EI(r_3,\#(\langle\alpha,\beta\rangle)))) : \exists z C(x,y,z)$$

where t_1,t_2 are the construction notations for Π_1, Π_2.

Now, consider the result of specializing the construction f to a particular value for y; say $y = r_0$ where r_0 is a closed term of L. The specialized construction may be written,

$$\lambda x.\{f(x,r_0)\} = \lambda x.\ OE(\alpha,t_1,EI(r_1,\#(\alpha)) \quad OE(\beta,t_2[y \leftarrow r_0],EI(r_2,\#(\beta)), \quad EI(r_3,\#(\langle\alpha,\beta\rangle)))) : \exists z C(x,r_0,z)$$

Suppose that the normal form of $t_2[y \leftarrow r_0]$ is $OI_1(t_3)$ where t_3 does not contain α free - that is to say, suppose that t_2 when normalized returns the decision that $F(r_0)$ and not $G(r_0)$

holds, and also that the proof of this decision does not use the assumption $B(x)$. Then the result of normalizing $\lambda x.\{f(x,r_0)\}$ without pruning is,

(a) $\lambda x.\ OE(\alpha,t_1,EI(r_1,\#(\alpha)),EI(r_2,\#(t_3))):\exists zC(x,r_0,z)$,

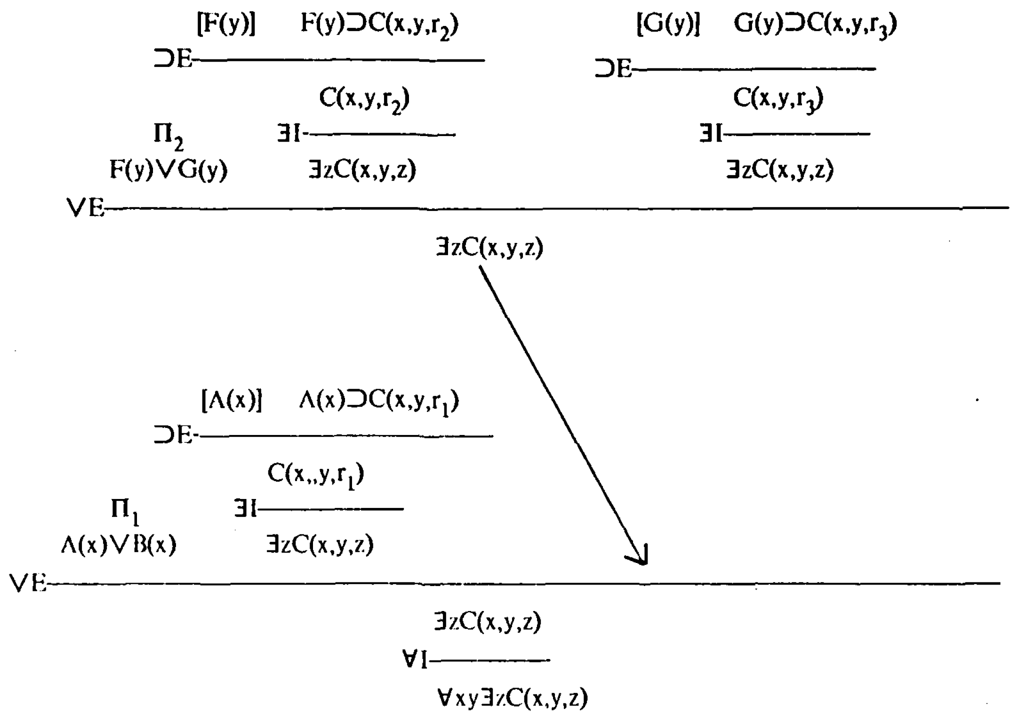
Pruning can be applied to the above expression, yielding

(b) $EI(r_2,\#(t_3)):\exists zC(x,r_0,z)$

Now, suppose that t_1 represents an extremely slow algorithm, so that t_1 applied to any particular argument takes a long time to normalize. Then the passage from (a) to (b) represents a large increase in efficiency: the normalization (without pruning) of the construction (a) on an input r requires that $t_1[x \leftarrow r]$ be normalized, whereas the construction (b) supplies the output "r_2" for all inputs, and so requires no reduction steps at all.

How slow can be t_1 be? The answer, for all practical purposes, is, arbitrarily slow, since recursive constructions can "run" as slowly as any recursive function. Even if t_1 is a recursion-free construction, normalization can still take so long as to be completely infeasible. In particular, there is no elementary recursive function in n which bounds the numer of reduction steps required to normalize non-recursive constructions of size n [Statman 1977]. (In other words, there is no such bounu of the form 2^n, or of the form $2^{(2^n)}$, or of the form $2^{(2^{(2^n)})}$, and so on).

So, we have demonstrated point (1) above. Point (2) can be demonstrated using a similar schematic example. Consider the following proof.



The above proof differs from the first proof P_1 only in that "$C(x,y,r_3)$" no longer depends upon "$B(x)$". The construction notation for P_2 is:

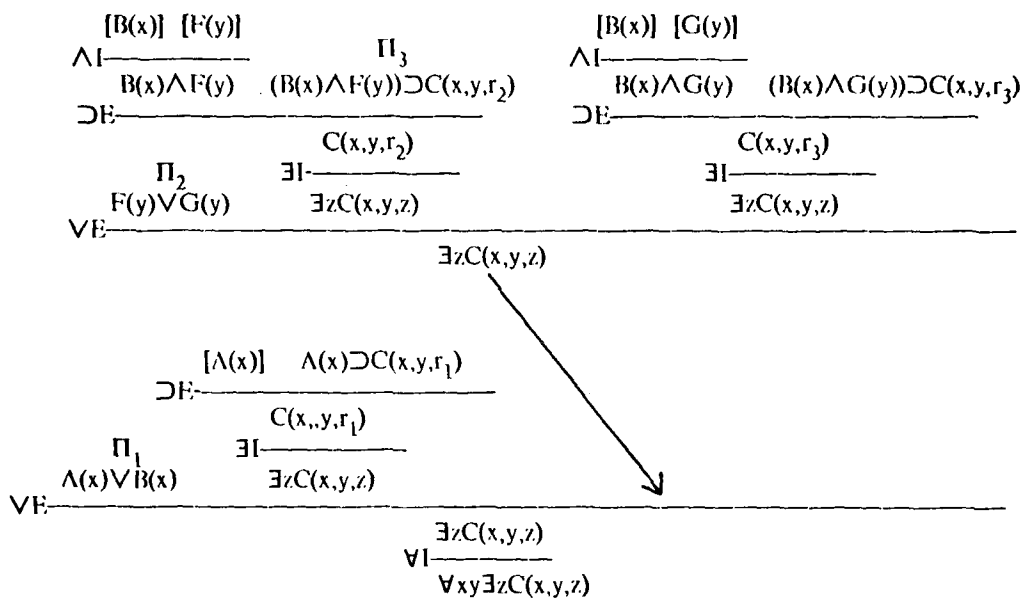
$$
\begin{aligned}
g \quad = \quad & \lambda xy.\ OE(\alpha,t_1,EI(r_1,\#(\alpha)) \\
& \qquad OE(\beta,t_2,EI(r_2,\#(\beta)), \\
& \qquad\qquad EI(r_3,\#(\beta)))): \exists zC(x,y,z)
\end{aligned}
$$

If pruning is applied to g, one gets

$$
\begin{aligned}
& \lambda x\ y.\ OE(\beta,t_2,EI(r_2,\#(\beta)), \\
& \qquad EI(r_3,\#(\beta))): \exists zC(x,y,z)
\end{aligned}
$$

Now, suppose in this case that t_1 is a fast algorithm, that is, that $t_1[x \leftarrow r]$ can be normalized in just a few steps for each input r. Suppose further that t_2 is very slow. Then we have the following situation: whenever $A(x)$ holds, r_1 may be immediately returned as the output, but when $B(x)$ holds a long computation must be undertaken to determine which of $F(y)$ and $G(y)$ holds. However, the correctness of the "long computation" does not depend on whether $B(x)$ holds. Thus we have a fast way (t_1) of discriminating between two ways of computing a satisfactory output, one of which is very fast (the simple return of r_1), and the other of which is very slow ("$\lambda x\ y.\ OE(\beta, t_2, EI(r_2, \#(\beta)), EI(r_3, \#(\beta))) : \exists z C(x,y,z)$"). Further, the slow way always works. Pruning has the effect of throwing away the discrimination (t_1) and choosing the slow way every time. Evidently, if $A(x)$ holds for many values of x, then pruning degrades the average efficiency of the algorithm. In the extreme case where $A(x)$ holds for all x, pruning takes a very fast algorithm and replaces it by a very slow one.

Point (2) has now been demonstrated, and we turn to point (3). As we have seen, all proofs of Harrop formulas may be omitted without interfering with the possibility of "running" a proof or construction. However, we will show here that the inclusion of a proof of a Harrop formula can extend the possiblities for pruning. As a consequence, the inclusion of proofs of Harrop formulas can in some cases improve the effectiveness of pruning in optimizing algorithms. We consider a third minor variant of the original schematic proof P_1:



In this case the change from P_1 is that $C(x,y,r_2)$ now appears to depend on both $B(x)$ and $F(y)$. The contruction notation for this proof is,

$$
\begin{aligned}
h \;=\; \lambda xy.\ & OE(\alpha, t_1, EI(r_1, \#(\alpha)) \\
& \qquad OE(\beta, t_2, EI(r_2, t_3(\langle\alpha,\beta\rangle)), \\
& \qquad\qquad EI(r_3, \#(\langle\alpha,\beta\rangle)))) : \exists z C(x,y,z)
\end{aligned}
$$

We assume for the current discusion that C is a Harrop formula. Thus "$(B(x)\wedge F(y))\supset C(x,y,r_2)$" is a Harrop formula, and could have been given simply as an axiom. Suppose however that the proof Π_3 of "$(B(x)\wedge F(y))\supset C(x,y,r_2)$" has the form:

$$\vee E\,\dfrac{\begin{array}{c}\Pi_4\\ H(y)\vee I(y)\end{array} \qquad \supset E\,\dfrac{\wedge I\,\dfrac{[F(y)]\;\;[H(y)]}{F(y)\wedge H(y)} \quad (F(y)\wedge H(y))\supset C(x,y,r_2)}{C(x,y,r_2)} \qquad \supset E\,\dfrac{\wedge I\,\dfrac{[B(x)] \qquad \wedge I\,\dfrac{[F(y)]\;\;[I(y)]}{F(y)\wedge I(y)}}{B(x)\wedge F(y)\wedge I(y)} \quad (B(x)\wedge F(y)\wedge I(y))\supset C(x,y,r_2)}{C(x,y,r_2)}}{C(x,y,r_2)}$$

Thus $C(x,y,r_2)$ may or may not actually depend on $B(x)$; if $H(y)$ holds it doesn't, and if $I(y)$ holds it does. The construction notation t_3 for the above proof is,

$$OE(\gamma,t_4,\#(\langle\beta,\gamma\rangle),\#(\langle\alpha,\langle\beta,\gamma\rangle\rangle)):C(x,y,r_2)$$

So, h has the form:

$$\begin{aligned} h \;=\; \lambda xy.\; & OE(\alpha,t_1,EI(r_1,\#(\alpha)) \\ & OE(\beta,t_2,EI(r_2,OE(\gamma,t_4,\#(\langle\beta,\gamma\rangle),\#(\langle\alpha,\langle\beta,\gamma\rangle\rangle))), \\ & EI(r_3,\#(\langle\alpha,\beta\rangle)))):\exists zC(x,y,z) \end{aligned}$$

Suppose that h is specialized to $\lambda x.h(x,r_0)$, where $H(r_0)$ and $F(r_0)$ hold (according to normalization of t_2 and t_4, which yield $OI_1(t_4)$ and $OI_1(t_5)$ respectively; we assume that neither t_4 nor t_5 contains α free). Then if $\lambda x.h(x,r_0)$ is normalized without pruning the following construction results.

$$\begin{aligned} \lambda x.\; & OE(\alpha,t_1,EI(r_1,\#(\alpha)) \\ & EI(r_2,\#(\langle t_4,t_5\rangle)) \end{aligned}$$

Finally, pruning yields,

$$EI(r_2,\#(\langle t_4,t_5\rangle)$$

Evidently, if the proof Π_3 for $C(x,y,r_2)$ had not been given, there would have been no possiblility of applying this last pruning operation. By the same argument given above for point (1), this pruning can lead to a large increase of efficiency.

Thus, although proofs of Harrop formulas are not required for the execution of a proof, they can be used to improve the analysis of dependencies upon which pruning relies.

Chapter 4

Specialization of a Bin-Packing Algorithm

The experiments described in this chapter demonstrate that prunable redundancies occur in the "real" computational world. The experiments concern the specialization of the first-fit backtracking algorithm for one-dimensional bin-packing. This algorithm takes a list L_1 of block sizes and a list L_2 of bin sizes as input. Each block and bin is "one-dimensional" in the sense that its size is given by a single positive number. The algorithm performs a depth-first search for a packing of the blocks into the bins - that is, for an assignment of the blocks to the bins with the property that the sum of the sizes of the blocks assigned to any given bin is less than the size of that bin. If such an assignment is found, the algorithm returns that assignment as its result, and otherwise it returns an indication that no packing exists. The algorithm is referred to as a "first fit" algorithm because, in the course of search, it attempts to place a block in the first bin in which it fits as its initial try. The bin-packing problem is well known to be NP-complete [Garey and Johnson, 1979], and this particular algorithm has a worst case running time which is exponential in the size of the input. However, the problem is tractable for small inputs. It is of interest to see how much the algorithm can be sped up in the cases where the inputs are of feasible size.

The bin-packing algorithm was formalized as a natural deduction proof in the first order theory of lists and numbers, and an untyped p-calculus term was extracted from this proof. The proof was constructed "by hand", but the extraction of the p-term from the proof, and all other phases of the experiments, were carried out automatically by a system of proof manipulation programs running on the Stanford Artificial Intelligence Laboratory PDP-10 computer. Several experiments were carried out, each of which involved specializing the algorithm to handle problems of a particular size and structure. For example, a specialized algorithm for packing six blocks given in order of descending size into three bins of equal size was derived from the general bin-packing algorithm by the following steps. (1) The p-calculus term which describes the general algorithm was executed (normalized without pruning) on the symbolic inputs $L_1 = \langle\langle i_1,i_2,i_3,i_4,i_5,i_6 \rangle\rangle$, $L_2 = \langle\langle n,n,n \rangle\rangle$, where the i_j and n are numeric variables, and where it was assumed further that $i_1 \geq i_2 \geq \ldots \geq i_6$. The resulting p-calculus term had the form of a decision tree. (2) The decision tree was subjected to an optimization involving the elimination of case analyses whose outcome was decided by formulas already assumed on the branch so far taken in the tree. The optimization was carried out by use of the simplex algorithm (all the case analysis predicates in bin-packing have the form of inequalities between sums). The process so far could as easily have been carried out on an ordinary program as on a proof or p-calculus term. However, at stage (3) pruning was applied. The question of central interest was this: what increase in speed and reduction in size would be obtained by the application of pruning?

In practice, it was not feasible to carry out steps 1 and 2 separately, since the decision tree resulting from step 1 would have been extremely large. Instead, normalization and optimization were applied "in parallel" - the decision tree was optimized in the course of its construction.

Experiments of the kind just described were carried out for all combinations of numbers n_1 of blocks and numbers n_2 of bins with $2 \leq n_2 < n_1 \leq 6$. In all cases, pruning turned out to be a useful optimization. As an example, we consider again the case where $n_1 = 6$ and $n_2 = 3$. The decision tree which results from steps (1) and (2) has 87 decision nodes and a depth of 14. When pruning is applied, the tree shrinks to 15 decision nodes with a depth of 8. Thus more than 4/5 of the decision nodes in the decision tree resulting from steps (1) and (2) are redundant in the sense recognized by pruning. If one measures the running time of a bin-packing algorithm by the number of comparisons which it makes, then the worst case running time of the original algorithm on inputs of the special form currently under consideration is 174. The worst case running time of a decision tree algorithm according to this measure is simply the depth of the tree. Thus the simplex optimization and pruning taken together produce a factor of improvement of nearly 22 in worst case running time (from 174 to 8).

As mentioned in section 2.8, pruning may have the effect of changing the function computed by a proof. Pruning does in fact have this effect in each of the experiments described in this chapter. Furthermore, this effect is essential to the success of pruning in improving efficiency. For $2 \leq n_2 < n_1 \leq 4$, the algorithm produced by pruning (*in combination* with symbolic execution and the simplex optimization) is both smaller and faster than any decision tree algorithm which *computes the same function as the original algorithm.* (This is may be true for $n_1 = 5$ and $n_1 = 6$ as well, although this has not been checked.) Thus, no collection of conventional optimizations could have produced specialized algorithms for bin-packing which are as efficient as those produced by pruning, since conventional optimizations preserve the extensional meaning of the programs to which they are applied.

The following conclusions can be drawn from the experiments. (1) The simplex optimization with or without pruning yields a large speed-up of the algorithm. (2) Pruning dramatically decreases the size of the specialized decsion tree algorithm, and produces a moderate improvement in its speed (ie depth). (3) The improvements produced by pruning could not have been produced by conventional optimizations. In the largest experiments (where $n_1 = 6$ and $n_2 \geq 4$), it was not feasible to produce a decision tree algorithm at all without the use of pruning; pruning had to be run in parallel with the simplex optimization and normalization in order to avoid running out of memory space. Thus in this application, the main practical effect of pruning was to make possible the production of fast specialized algorithms which are of a reasonable size. In devising combinatorial algorithms for handling a finite number of cases, speed is not the only problem, since one can often make use of table look-up to get a very fast but very large algorithm. What is difficult is to produce an algorithm which is *both* small and fast.

In what follows, we describe the experiments in detail. Section 4.1 concerns the system of programs used for doing the experiments. In section 4.2, we describe the proof which implements the bin-packing algorithm. Section 4.3 concerns the reductions on object terms used in normalization. Section 4.4 gives the results of the experiments. Conclusions based on the results are given briefly in section 4.5.

4.1 The implementation

The system of programs used for the experiments was written in MacLISP, and runs on the Stanford Artificial Intelligence Laboratory PDP-10 computer. The system consists of three components: (1) a proof checker for natural deduction, (2) a mechanism for extracting untyped p-calculus terms from proofs, and (3) a normalizer (with pruning) for the p-calculus. The proof checker is interactive, and allows the user to specify the first-order language in which a proof is to be given. In these respects, it resembles the FOL proof checker [Weyhrauch 1974].

The normalizer, both in internal design and in function, is very much like interpreters for λ-calculus based languages such as LISP[McCarthy et al, 1962] and SCHEME[Sussman and Steele, 1975]. The execution of a LISP or SCHEME program is essentially a matter of normalizing a *closed* λ-calculus term which ends up with an object term as its normal form. In the case of SCHEME, where the static binding convention is observed, the interpreter has *exactly* the effect of a λ-calculus normalizer when applied to a closed term having a "concrete value", whereas in most standard dialects of LISP (eg LISP 1.6, MacLISP, InterLISP), dynamic binding holds sway, leading to a somewhat different behavior than normalization. In any case, there exists a well developed technology for efficient normalization of some kinds of λ-terms, and this technology is easily adapted to the task of normalization in the p-calculus.

A central element of this technology is the use of environments for implementing substitutions. The idea here is this. An environment is an association $\{(x_1,t_1),(x_2,t_2)\ .\ .\ .\ (x_n,t_n)\}$ of terms with variable names. If one wishes to evaluate (or normalize) a term which is given as the result of $t_1[x \leftarrow t_2]$ of a substitution, then, instead of doing the substitution first and the normalization afterwards, one normalizes the term t_1 *in the environment* $\{(x,t_2)\}$. The normalization of a term t in an environment e is like normalization of the usual kind, except that variables which have been assigned values in the environment are regarded as temporary names for those values. Most reduction rules applied in the course of normalizing t do not make use of the internal structure of the subterm which a temporary name designates; on occasions when this internal structure is relevant, the value assigned to the name is looked up. We won't go into further detail about how environments are used; the reader who is unfamiliar with these techniques should see [McCarthy et al, 1962].

Our normalizer uses environments in the implementation of $\forall$-reduction, $\supset$-reduction, $\vee$-reduction, and $\exists$-reduction. The normalizer resembles traditional interpreters in the additional respect that a "call-by-value" reduction order is used. That is to say, except for terms whose main constructor is APPLY, OE or EE, a term t with immediate subterms $t_1 \ldots t_n$ is normalized by first normalizing each t_i, and then applying reductions to the result. In the case of (1) APPLY(t_1,t_2, . . . t_n) , (2) OE(α,t_1,t_2,t_3), and (3) EE(x,α,t_1,t_2), t_1 is normalized first. If t_1 has the form (1)$\lambda\underline{v}$.t, (2) $OI_1(t)$ or $OI_2(t)$, (3) EI(t,t'), then (1) t, (2) t_2 or t_3, (3) t_2 is normalized in the extension of the current environment which associates (1) $t_2 \ldots t_n$ with $v_1, \ldots v_n$, (2) t with α, (3) x with t and α with t'. If t_1 does not have the appropriate form to allow a reduction rule to be applied, then $t_2 \ldots t_n$ are normalized in sequence.

The normalizer is an iterative program in the style of the SCHEME interpreter [Sussman & Steele 1975]. A collection of (software) switches controls the mode in which the normalizer operates. For example, the pruning reductions and the permutation operations can be turned on and off at will. Proof procedures (section 2.5) are implemented by calls from the normalizer to ordinary LISP functions. The entire system, including the proof checker, the extractor, the normalizer, and a top level, constitutes about 900 lines of MacLISP code, and when compliled occupies 70,000 words 36-bit words of memory. The former figure includes only the code which was written by the current author specifically for the proof manipulation system. It does not include the code contained in the two "packages" which were imported into system, namely a general purpose pretty-printer written by Derek Oppen (see [Oppen, 1979]) and a simplex algorithm written by Greg Nelson. The figure of 70,000 words, however, measures the total amount by which the size of the proof manipulation system exceeds that of "bare" MacLISP.

4.2 The proof

The bin-packing proof used in the experiments is formulated in a first order language L_0 for numbers and lists of numbers. There are two sorts of variables: variables which range over non-negative integers, and variables which range over lists of non-negative integers. (Note that the use of sorted variables where the sorts are disjoint has no effect whatever on the treatment of proofs as computational descriptions; in normalization and the extraction of p-terms, the sort information may be simply ignored.) In what follows, lower case letters are used for numeric variables, while capital letters are used for variables which range over lists.

The function and relation symbols of L_0 are listed below, together with their intended meanings. Note that some of the symbols are given as infix operators. Any language definition supplied to the proof checker includes information as to which binary function and relation symbols are to be treated as infix operators by the parser for formulas and terms. Our usage below directly reflects this syntactic part of the formal defintion of L_0.

symbol	intended meaning
+	n+m is the sum of n and m.
–	n-m is the result of subtracting m from n.
<	n<*m* holds *if n* is less *than m.*
≤	n≤m holds if n is less than or equal to m.
lnth	lnth(A) is the length of the list A
@	n @ A is the list which results from adding n to the front of A
:	A:n is the nth element of A (it makes no difference for our purposes how A:n is defined for n=0 or n > lnth(A))
tl	tl(A) (read "tail of A") is the result of removing the first element from the list A; the tail of the empty list is the empty list.
set	set(A,n,m) is the list which results from replacing the nth element of A by m. If n=0 or n > lnth(A) then set(A,n,m) is A.

It is most convenient to think of L_0 as having just one list constant, namely "≪≫" for the empty list, and infinitely many numeric constant symbols: one for each number. The numeric constants (numerals) are represented in a direct fashion in the implementation, namely by

LISP numbers. The parser and the programs which print out proofs and p-terms use ordinary decimal notation for numerals.

We will abbreviate a term having the form "t_1 @ t_2 @ t_3 @ . . . t_n @ $\ll\gg$" by "$\ll t_1, t_2, \ldots t_n \gg$". Thus t_i in $\ll t_1, t_2, \ldots t_n \gg$ denotes the ith element of the list denoted by $\ll t_1, t_2, \ldots t_n \gg$. The parser and the output programs also use this notation (in fact, the same kind of abbreviation is used in the internal representation of terms). Also, "null(X)" will serve as an abbreviation for "$X = \ll\gg$".

We can state the one-dimensional bin-packing problem in the following way. Suppose that we have n blocks and m bins. Each block and each bin has a particular *size* given by a positive integer. Let $X = \ll i_1, \ldots i_p \gg$ be a list of the sizes of the blocks, and let $B = \ll j_1, \ldots j_q \gg$ be a list of the sizes of the bins. An assignment of the blocks X to the bins B will be represented by a list $\ll k_1, \ldots k_p \gg$, where k_m is the number of the bin to which the mth block is assigned. For example, {2,1,1} represents an assignment of three blocks to two bins, where the first block is assigned to the second bin, and the remaining two blocks are assigned to the first bin.

Now, an assignment $A = \ll k_1, \ldots k_p \gg$ of blocks X to bins B is *legal* if each block of X is assigned to some bin of B (ie if lnth(A)=lnth(X), and $k_m \leq$ lnth(B) for each m), and if the sum of the sizes of the blocks assigned to any one bin is less than or equal to the size of that bin. The one-dimensional bin-packing problem is this: given lists of block sizes X and bin sizes Y, determine whether there is a legal assignment A of the blocks to the bins, and if there is, give it.

The algorithm for bin-packing which is used in the experiments is as follows, expressed as an ordinary definition by mutual recursion.

```
pack(X,B) ← if null(X) then ≪≫ else packb(X,B,1)

packb(X,B,n) ← if n≤lnth(B) then
                   if X:1≤B:n then
                      if pack(tl(X),set(B,n,B:(n - X:1)))≠FAIL then
                           (n @ pack(tl(X),set(B,n,B:(n - X:1))))
                           else packb(X,B,n+1)
                      else FAIL
                   else FAIL
```

An informal explanation of the workings of this algorithm is as follows.

The function *pack* takes a list of blocks X and bins B and returns either a legal assignment of the blocks to the bins, or "FAIL", meaning that there is no packing. *Pack* first checks whether there are any blocks (ie whether X is null). If not, then the null assignment will do. Otherwise, the function *packb* is called with the bound n set to 1. In *packb*(X,B,n) it may be assumed that X is non-empty.

The "bounded" packing function *packb* attempts to find a packing of the blocks X into the bins B subject to a restriction on where the first block in X may be put; namely, the first block must be assigned to a bin whose index is n or greater. *Packb* first checks whether n is greater than the length of B; if this is the case then no packing which satisfies the given restriction is possible. Otherwise, *packb* checks whether the first block fits in the nth bin (ie whether $X{:}1 \leq B{:}n$). If the block fits, then an attempt is made to pack the rest of the blocks into the space which remains in the bins; specifically *pack*(tl(X),B′) is called, where B′ = set(B,n,B:(n - X:1)). B′ differs from B in that the size of the nth bin has been reduced to reflect the assignment of the first block to that bin. If such a packing A of tl(X) into B′ is found, then n @ A evidently suffices as a packing of X into B. Finally, if no packing of tl(X) into B′ is possible, or if X:1 did not fit into B:n in the first place, then *packb*(X,B,n+1) is called. Thus, the end effect executing *packb*(X,B,1) is that the first block X:1 is placed succesively in the first bin in which it fits, the second bin in which it fits, and so on, until a placement of X:1 is found which can be extended to a complete packing of X into B, or until no bins are left.

Note that there are two identical calls to *pack* in the body of *packb*. This duplication of effort could easily have been eliminated by the use of a λ-abstraction, but this was not done for the sake of simplicity of presentation. The duplication does not appear in the bin-packing proof.

The bin-packing proof has two parts: a "main theorem" PACK, and a lemma PACKB. Formally, "PACK" and "PACKB" are to be regarded as defined symbols of the language L_0, to which "recursive" proofs have been assigned according to the rules given in section 3.4. These proofs correspond closely in structure to the recursive definitions *pack* and *packb*; the proofs embody the same analysis of cases, and the same pattern of recursive calls - in short, the same algorithm - as do *pack* and *packb*. Listings of PACK and PACKB are given below in the form in which they were printed out by the proof checker. The notation used by the proof checker is somewhat unusual and will be explained shortly. But first, here is the listing of PACK.

1	NULLD(X)	null(X)∨(¬null(X))	
2	AS(null(X))	null(X)	2
3	AX(∀B(legal(≪≫,≪≫,B)))	∀B(legal(≪≫,≪≫,B))	
4	EV(*2)	X=≪≫	2
5	SB(*4,*3(B),2)	legal(≪≫,X,B)	2
6	EI(≪≫,*5,∃A(legal(A,X,B)))	∃A(legal(A,X,B))	2
7	AS(¬null(X))	¬null(X)	7
8	PACKB(X,B,1)(*7)	∃A(BLA(A,X,B,1))∨(¬∃A(BLA(A,X,B,1)))	7
9	AS(∃A(BLA(A,X,B,1)))	∃A(BLA(A,X,B,1))	9
10	EV(*9)	∃A(legal(A,X,B)∧(¬null(X))∧(1≤(A:1)))	9
11	AS(legal(A,X,B)∧(¬null(X))∧(1≤(A:1)))		
		legal(A,X,B)∧(¬null(X))∧(1≤(A:1))	11
12	[*11↓1]	legal(A,X,B)	11
13	EI(A,*12,∃A(legal(A,X,B)))	∃A(legal(A,X,B))	11
14	EE(*10,*13,A)	∃A(legal(A,X,B))	9
15	OI(*14,¬∃A(legal(A,X,B)))	∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B)))	9
1	AS(¬∃A(BLA(A,X,B,1)))	¬∃A(BLA(A,X,B,1))	1
17	AX(∀X B(¬null(X),¬∃A(BLA(A,X,B,1))→¬∃A(legal(Aα3,X,B))))(X,B)(*7,*1)		
		¬∃A(legal(Aα3,X,B))	7,1
18	OI(∃A(legal(A,X,B)),*17)	∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B)))	7,1
19	OE(*8,*15,*18)	∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B)))	7
20	OI(*,¬∃A(legal(A,X,B)))	∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B)))	2
21	OE(*1,*20,*19)	∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B)))	
22	λX B(*21)	∀X B(∃A(legal(A,X,B))∨(¬∃A(legal(A,X,B))))	

First we comment on two predicate symbols which appear in the above listing. The formula legal(A,X,B) holds if A is a legal assignment of the blocks X to the bins B. WIth a bit of work, "legal" can be defined from the primitive operators and predicates of L_0 which were given above, but there is no reason to do so here. For the current purposes, "legal" is treated as primitive. The formula BLA(A,X,B,n) holds if A is a "legal bounded assignment" of the kind that *packb* might generate - that is to say a legal assignment of a non-empty list X of blocks to bins B which assigns the first block to a bin whose index is at least n. BLA is used as a defined predicate; its definition is

BLA(A,X,B,n) = legal(A,X,B)∧(¬null(X))∧(n≤(A:1))

The above listing should be regarded as a "linear" notation for a natural deduction proof tree of the kind discussed in chapter 2. Each line of the listing designates a node of the tree. A line has four pieces of information associated with it. Reading from right to left, these are (1) the line number, (2) a term, (3) a formula, and (4) a list of dependencies. The line numbers serve simply as unique labels which are used to identify the line in question. The term indicates the the sequence of inferences by which the current line was arrived at from previous lines of the proof. The formula is simply the formula associated with the node in the proof tree which the line designates; it is the conclusion of the inferences which have been completed thus far. Finally, the list of dependencies is a list of the line numbers of the assumptions upon which the conclusion of the current line depends. In the interactive construction of a proof, the user types a term of the kind suitable for the term part (2) of a proof line; the proof checker then assigns a new line number and computes the formula and dependencies of the new line.

This method of laying out a proof tree in linear fashion is of course quite standard. The only unusual aspect of the notation is the manner in which the application of inference rules is described. This information, as we have said, appears in the form of the term which is the second part of every proof step or line; this term resembles a p-term in several ways, and will be called a "q-term". A q-term is built up from axioms and assumptions and from references to previous lines by the application of operators which represent inference rules. An axiom is is given by a q-term of the form "AX(φ)" where φ is the formula being asserted as an axiom (see line 3), while an assumption has the form AS(φ) (see line 2). References to previous proof steps take the form of an asterisk followed by the line number of the step. The operators which represent inference rules are: PAIR for $\wedge$-introduction, UNPAIR for $\wedge$-elimination, OI for $\vee$-introduction, OE for $\vee$-elimination, II for $\supset$-introduction, APPLY for $\supset$-elimination and $\forall$-elimination, λ abstraction for $\forall$-introduction, EI for $\exists$-introduction, and finally SB for substitution (of "right for left"). The syntax of q-terms is largely borrowed from the syntax which we have been using for p-terms; for example "PAIR(t_1,t_2)" is written "<t_1,t_2>". Q-terms differ from p-terms in the significant aspect that no proof variables are used; a q-term is no more than a fragment of an ordinary natural deduction proof written in applicative syntax.

As a simple example of a q-term, consider the term part of line 19 of the proof PACK, which reads "OE(*1,*20,*19)". This designates the result of applying $\vee$-elimination to the premises represented by lines 1,20, and 19 repectively. A more complicated example is the term part of line 8. As usual, we use the syntax "t_1(t_2)" for APPLY(t_1,t_2). APPLY, in turn is used to designate both the $\forall$ elimination and $\supset$-elimination inference rules. Thus a q-term of the form t(t_1, . . . t_n) designates either an $\forall$ elimination or an $\supset$-elimination rule, under the condition that t is not itself an inference rule name. Now, the term part of line 8 is

"PACKB(X,B,1)(*7)". PACKB is the lemma which corresponds to the bounded packing function "*packb*". The endformula of PACKB is

$$\forall X\ B\ n(\neg null(X) \supset \exists A(BLA(A,X,B,1)) \vee (\neg \exists A(BLA(A,X,B,1))))$$

The formula PACKB(X,B,1) designates the result of an $\forall$-elimination with PACKB as the premise, followed by an $\supset$-elimination with line 7 as the minor premise. Thus two inference rule applications are described by line 8 of the proof. In general, one can record as many inferences as one desires in a single line of proof by the use of a suitably complicated q-term; the decision as to how much information is to be included in each line is a matter of convenience.

What we have said so far should make at least a rough understanding of the proof PACK possible. An informal outline of the proof is as follows. First of all, the proof takes the form of a case analysis according to whether X is null (see steps 1 and 20). Steps 2 through 6, and step 20, take care of the case where X is null. If X is not null, the lemma PACKB is used (step 8). Steps 9 through 19 are devoted to showing that

$$\exists A(legal(A,X,B)) \vee (\neg \exists A(legal(A,X,B)))$$

can be derived from

$$\exists A(BLA(A,X,B,1)) \vee (\neg \exists A(BLA(A,X,B,1)))$$

This is done by a case analysis (step 19) according to whether $\exists A(BLA(A,X,B,1))$ is true. The outer case analysis of PACK - namely the case analysis according to whether X is null - is reflected directly by the conditional expression "if null(X) then $\ll\gg$ else *packb*(X,B,1)" in the ordinary recursive defintion *pack*. However, the inner case analysis which has just been mentioned is necessary only in order to demonstrate that the value returned by *packb*(X,B,1) is also a valid output for *pack*; no counterpart of this case analysis is present in the ordinary recursive defintion.

Further information concerning the notation used by the proof checker, and concerning the proofs PACK and PACKB, is given below. None of this information is of any general significance; our current purpose is to provide the detail necessary for a full step-by-step understanding of the proofs PACK and PACKB.

° One lemma other than PACKB appears in PACK, namely NULLD (line 1). The "endformula" of NULLD is $\forall X(null(X) \vee \neg null(X))$. A proof procedure for NULLD is supplied as part of the normalizer; NULLD(t) returns $OI_1(\#)$ if t is "$\ll\gg$", and $OI_2(\#)$ if t has the form "$\ll t_1, \ldots t_n \gg$" where $n \geq 1_n$. Also, the lemma LTED appears in PACKB. The endformula of LTED is "$\forall n\ m(n \leq m \vee m < n)$"; the proof procedure LTED(t_1,t_2) returns $OI_1(\#)$ if t_1,t_2 are numerals with $t_1 \leq t_2$, and $OI_2(\#)$ if t_1,t_2 are numerals with $t_2 < t_1$.

° The operator "EV" has the effect of removing abbreviations in the endformula of a proof - that is, of replacing defined predicates by their definitions. Two defined predicate symbols appear in PACK, namely "null" and "BLA". These symbols are removed by EV in lines 4 and 10, respectively. EV should not be thought of as an inference rule, but rather as part of a facility in the proof checker which allows formulas to be given in an abbreviated notation; from this point of view, EV has the effect of changing the external form in which a formula is presented to the user without changing the formula itself. Evidently, uses of EV could be dispensed with in any proof simply by replacing all abbreviations by their definitions throughout the proof. The procedure which extracts p-terms from proofs ignores uses of EV; that is to say, the term which is extracted from "EV(Π)" is the just the term extracted from "Π". Similarly, the operator "EVQ", which appears in PACKB but not in PACK, is used in conjunction with SB to *introduce* abbreviations. EVQ is applied to a formula rather than a proof; EVQ(φ) produces a proof step whose "formula" part is "$\varphi = \psi$", where ψ is the formula which results from removing the abbreviations from φ. However, "$\varphi = \psi$" should not be regarded as a formula but rather as another artifact of the abbreviation facility. The operator SB may be used with "$\varphi = \psi$" as its first premise in order to substitute the abbreviated form φ for the expanded form ψ in the enformula of its second premise. EVQ and SB are used together in this manner in steps 14 and 15, and steps 28 and 29, of PACKB. Again, these steps could be removed by replacing all abbreviations by their definitions throughout the proof.

° There are two variants of the $\vee$-introduction inference - one puts the "new" disjunct on the right, and the other puts it on the left. The corresponding forms of an application of the "OI" operator are: (a) OI(Π,F), and (b) OI(F,Π), where Π is a proof, and F is a formula (F is the "new" disjunct). More explicitly, let us suppose that the endformula of Π is A. Then the endformulas of the proofs which result from the two forms (a) and (b) of OI will be $A \vee F$, and $F \vee A$, respectively.

° An application of the "EI" operator for $\exists$-introduction has the form "EI(t,Π,$\exists x \varphi$), where t is a term of L, Π is a proof, and $\exists x \varphi$ is (of course) a formula. It is assumed that the endformula of Π has the form $\varphi[x \leftarrow t]$; otherwise the proof checker will reject this application of EI. $\exists x \varphi$ is the endformula of the result of the application.

° In PACK and PACKB we make use of the connectives "$\wedge$" and "$\supset$" as operators of arbitrary arity. That is to say, just as we have allowed "$\forall$" to quantify over not just one, but arbitrarily many variables, we allow formulas of the forms $[A_1, A_2, \ldots A_n \supset B]$, and of the form $[A_1 \wedge A_2 \wedge \ldots A_n]$, where each of these is to be regarded as the result of applying a single high arity connective "$\supset$" or "$\wedge$" to $A_1, \ldots A_n$ and in the first case B. The meaning of $[A_1, A_2 \ldots A_n \supset B]$ is just $(A_1 \wedge A_2 \wedge \ldots A_n \supset B)$. The inference rules which treat $\wedge$ and $\supset$ are modified in a suitable way. Namely, $\wedge$-introduction now takes as many premises as desired and produces the conjunction of all the premises as its conclusion. Correspondingly, one needs a separate variant of $\wedge$-elimination for selecting each of the conjuncts of a high arity conjunction. The q-term notation for $\wedge$-introduction is "$<\Pi_1, \ldots \Pi_n>$". For $\wedge$-elimination, we have "$[\Pi\downarrow 1]$" to select the first conjunct, "$[\Pi\downarrow 2]$" to select the second, "$[\Pi\downarrow 3]$" to select the third, and so forth. ("$[\Pi\downarrow k]$" corresponds to "π_k" in the notation which we have been using for p-terms). $\supset$-elimination also takes as many arguments as are appropriate to its major premise; in q-term notation we write "$\Pi(\Pi_1, \Pi_2, \ldots \Pi_n)$" to designate the application of $\supset$-elimination to the the major premise Π, and minor premises $\Pi_1, \Pi_2 \ldots \Pi_n$. It is assumed here that the endformula of Π has the form $[A_1, A_2 \ldots A_n \supset B]$, where $A_1, A_2 \ldots A_n$ are the endformulas of $\Pi_1, \Pi_2 \ldots \Pi_n$, respectively. The conclusion of this $\supset$-elimination inference is B. For an example of the use of $\supset$-elimination of arity 2, see step 17 of PACK. The use of arbitrary arity connectives constitutes an inessential but convenient extension of notation.

A listing of the proof PACKB is as follows.

1	AS(¬null(X))	¬null(X)	1
2	LTED(n,lnth(B))	(n≤lnth(B))∨(lnth(B)<n)	
3	AS(n≤lnth(B))	n≤lnth(B)	3
4	LTED(X:1,B:n)	(X:1≤(B:n))∨(B:n<(X:1))	
5	AS(X:1≤(B:n))	X:1≤(B:n)	5
6	PACK(tl(X),set(B,n,B:n−(X:1)))	∃A(legal(A,tl(X),set(B,n,B:n−(X:1))))∨ (¬∃A(legal(A,tl(X),set(B,n,B:n−(X:1)))))	
7	AS(∃A(legal(A,tl(X),set(B,n,B:n−(X:1)))))	∃A(legal(A,tl(X),set(B,n,B:n−(X:1))))	7
8	AS(legal(A,tl(X),set(B,n,B:n−(X:1))))	legal(A,tl(X),set(B,n,B:n−(X:1)))	8
9	AX(∀A X B n([¬null(X),n≤lnth(B),X:1≤(B:n), legal(A,tl(X),set(B,n,B:n−(X:1))) ⊃ legal(n@A,X,B)])	∀A X B n([¬null(X),n≤lnth(B),X:1≤(B:n), legal(A,tl(X),set(B,n,B:n−(X:1))) ⊃ legal(n@A,X,B)])	
10	*9(A,X,B,n)(*1,*3,*5,*8)	legal(n@A,X,B)	1,3,5,8
11	AX(∀n A(n≤(n@A:1)))	∀n A(n≤(n@A:1))	

```
12   *11(n,A)                     n≤(n@A:1)
13   <*10,*1,*12>                 legal(n@A,X,B)∧(¬null(X))∧(n≤(n@A:1))
                                                                          8,5,3,1
14   EVQ(BLA(n@A,X,B,n))
                                  BLA(n@A,X,B,n)=
                                     (legal(n@A,X,B)∧(¬null(X))∧(n≤(n@A:1)))
15   SBL(*14,*13)                 BLA(n@A,X,B,n)                          8,5,3,1
16   EI(<n@A>,*15,∃A(BLA(A,X,B,n)))
                                  ∃A(BLA(A,X,B,n))                        8,5,3,1
17   FE(*7,*16)                   ∃A(BLA(A,X,B,n))                        5,3,1,7
18   OI(*17,¬∃A(BLA(A,X,B,n)))
                                  ∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))
                                                                          5,3,1,7
19   AS(¬∃A(legal(A,tl(X),set(B,n,B:n   (X:1)))))
                                  ¬∃A(legal(A,tl(X),set(B,n,B:n   (X:1))))   19
20   PACKB(X,B,n+1)
                                  ¬null(X)⊃
                                     ∃A(BLA(A,X,B,n+1))∨(¬∃A(BLA(A,X,B,n+1)))

21   *20(*1)                      ∃A(BLA(A,X,B,n+1))∨(¬∃A(BLA(A,X,B,n+1)))
                                                                          1
22   AS(∃A(BLA(A,X,B,n+1)))
                                  ∃A(BLA(A,X,B,n+1))                      22
23   AS(BLA(A,X,B,n+1))           BLA(A,X,B,n+1)                          23
24   EV(*23)                      legal(A,X,B)∧(¬null(X))∧(n+1≤(A:1))     23
25   AX(∀n m(n+1≤m⊃n≤m))
                                  ∀n m(n+1≤m⊃n≤m)
27   <[*24↓1],[*24↓2],*25(n,A:1)([*24↓3])>
                                  legal(A,X,B)∧(¬null(X))∧(n≤(A:1))       23
28   EVQ(BLA(A,X,B,n))            BLA(A,X,B,n)=
                                     (legal(A,X,B)∧(¬null(X))∧(n≤(A:1)))

29   SBL(*28,*27)                 BLA(A,X,B,n)                            23
30   EI(A,*29,∃A(BLA(A,X,B,n)))   ∃A(BLA(A,X,B,n))                        23
31   FE(*22,*30)                  ∃A(BLA(A,X,B,n))                        22
32   OI(*31,¬∃A(BLA(A,X,B,n)))
                                  ∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))
                                                                          22
35   AS(¬∃A(BLA(A,X,B,n+1)))
```

		¬∃A(BLA(A,X,B,n+1))	35
36	LEM0(X,B,n)(*19,*35)	¬∃A(BLA(A,X,B,n))	19,35
37	OI(∃A(BLA(A,X,B,n)),*36)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	19,35
38	OE(*21,*32,*37)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	1,19
39	OE(*6,*18,*38)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	5,3,1
40	AS(B:n<(X:1))	B:n<(X:1)	40
41	PACKB(X,B,n+1)	¬null(X)⊃ ∃A(BLA(A,X,B,n+1))∨(¬∃A(BLA(A,X,B,n+1)))	
42	*41(*1)	∃A(BLA(A,X,B,n+1))∨(¬∃A(BLA(A,X,B,n+1)))	1
43	AS(¬∃A(BLA(A,X,B,n+1)))	¬∃A(BLA(A,X,B,n+1))	43
44	LEM1(X,B,n)(*1,*40,*43)	¬∃A(BLA(A,X,B,n))	1,40,43
45	OI(∃A(BLA(A,X,B,n)),*44)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	1,40,43
46	OE(*42,*32,*45)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	1,40
47	OE(*4,*39,*46)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	3,1
48	AS(lnth(B)<n)	lnth(B)<n	48
49	LEM2(X,B,n)(*48)	¬∃A(BLA(A,X,B,n))	48
50	OI(∃A(BLA(A,X,B,n)),*49)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	48
51	OE(*2,*47,*50)	∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	1
52	II(¬null(X),*51)	¬null(X)⊃∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n)))	
53	λX B n(*52)	∀X B n(¬null(X)⊃ ∃A(BLA(A,X,B,n))∨(¬∃A(BLA(A,X,B,n))))	

4.3 Reduction rules for terms of L_0

The following special purpose reduction rules for terms of L_0 are provided as part of the normalizer (reductions on object terms were discussed in section 2.6). We will not belabor the distinction between numerals and numbers; for example we will allow ourselves to use the phrase, "the sum of t_1 and t_2", instead of the more precise phrase "the numeral which denotes the sum of the numbers which t_1 and t_2 denote" in the case where t_1 and t_2 are numerals. However, special variables, namely, a,b and c, will be used for numerals.

"a + b" $\Rightarrow$ "c", where c is the sum of a and b.

"a − b" $\Rightarrow$ "c", where c is the result of the indicated subtraction.

"lnth($\ll t_1,t_2, \ldots t_n \gg$)" $\Rightarrow$ t' where t' is the numeral for n.

tl($\ll t_1,t_2, \ldots t_n \gg$) $\Rightarrow$ $\ll t_2, \ldots t_n \gg$

$\ll t_1,t_2, \ldots t_n \gg$:a $\Rightarrow$ t_a, under the condition that $1 \leq a \leq n$.

set($\ll t_1,t_2, \ldots t_n \gg$,b,$t_0$) $\Rightarrow$ t', where one of the following conditions holds: (a) $1 \leq b \leq n$ and t' is the result of replacing t_b in $\ll t_1,t_2, \ldots t_n \gg$ by t_0, (b) b=0 or b > n, and t' is $\ll t_1,t_2, \ldots t_n \gg$.

For example, these reduction rules would have the effect of reducing the term "$\ll$3,4+5$\gg$:2" to the term "9". It is not hard to see that normalization of any term of L_0 with respect to these rules will terminate, and that the normalization of any *closed* term will yield either a numeral, or a term of the form $\ll t_1,t_2, \ldots t_n \gg$ where the t_i are numerals.

4.4 Results

The results of the experiments will be presented in several stages. The p-terms which were extracted from the proofs PACK and PACKB will be given in section 4.4.1. In section 4.4.2, the results of the smallest of the experiments are given in full detail, and the simplex optimizations are described. Section 4.4.3 presents the optimized algorithm for packing six blocks into three bins. Finally, section 4.4.4 tabulates the results of the remaining experiments.

4.4.1 P-terms

The following p-term was extracted from PACK:

ppack =

```
λ X B
 (OE (α2
   NULLD(X)
   OI(1,EI(SB(α2,#(B)),<<>>))
   (OE (α4)
     PACKB(X,B,1)(α2)
     OI(1,EE (α6 A) α4 EI([α6↓1],A))
     OI(2,#(X,B)(α2,α4))))
```

The notation used for p-terms in the implementation differs in several minor ways from the notation which we have found it convenient to use in our exposition of the p-calculus in chapter 3. (1) In the implementation, we write "OI(1,t)" and "OI(2,t)" instead of "$OI_1(t)$" and "$OI_2(t)$". (2) As explained in the last section, we now allow "pairing" operators of each positive arity; arbitrarily many terms $t_1 \ldots t_n$, can be "tupled" together into the term $\langle t_1 \ldots t_n \rangle$. Correspondingly, there is a projection operator π_k for each positive integer k. Instead of writing "$\pi_k(t)$" we write "[t↓k]". Note that k *must be a numeral*; [t↓1], [t↓2], . . . are to be regarded as notations for separate elementary operators of the p-calculus. ("↓" is not a function symbol!) We remark once more that the use of arbitrary arity tupling instead of iterated pairing is no more than a notational convenience. (3) The order in which arguments to the operator "EI" appear is reversed; a p-term "$EI(t_1,t_2)$" as expressed in the notation of chapter 3 is written as "$EI(t_2,t_1)$" in the notation of the implementation. Thus, in a construction "$EI(t_1,t_2):\exists x\varphi$" in the new notation, t_1 is the construction for $\varphi(t_2)$, and not the other way around. (4) The numbers which play the role of subscripts to variables appear simply to the right of the variable name rather than to the right and below the variable name. Thus, we write "i1", "i2", "α1", "α2" and so forth, instead of "i_1", "i_2", "α_1", "α_2". (The reason for this change is that the text of the various p-terms given below was derived directly from the output of the proof checking system; such output, for practical reasons, does not make use of subscripts. The output was produced in indented form by use of Derek Oppen's[1979] pretty-printer.)

The p-term extracted from PACKB is:

ppackb =

```
λ X B n
 (λ α2
  (OE (α4)
    LTED(n,lnth(B))
    (OE (α6)
      LTED(X:1,B:n)
      (OE (α8)
        PACK(tl(X),set(B,n,B:n-(X:1)))
        OI(1,
          EE (α10 Λ)
            α8
            EI(SB(#,
                 <#(Λ,X,B,n)(α2,α4,α6,α10),α2,
                  #(n,Λ)>),n @ Λ))
        (OE (α12)
          PACKB(X,B,n + 1)(α2)
          OI(1,
            EE (α14 Λ)
              α12
              EI(SB(#,
                   <[α14↓1],[α14↓2],
                    #(n,Λ:1)([α14↓3])>),Λ))
          OI(2,#(X,B,n)(α8,α12))))
      (OE (α16)
        PACKB(X,B,n + 1)(α2)
        OI(1,
          EE (α18 Λ)
            α16
            EI(SB(#,
                 <[α18↓1],[α18↓2],#(n,Λ:1)([α18↓3])>),
              Λ))
        OI(2,#(X,B,n)(α2,α6,α16))))
    OI(2,#(X,B,n)(α4))))
```

The system of lemmas ppack and ppackb has the termination property with respect to our call-by-value normalizer; this can be established by exactly the same kind of argument as would be used to establish the termination of the ordinary recursive functions *pack* and *packb*.

Let t_1 and t_2 be closed terms for lists. By theorem 3.1 of chapter 3, the result of normalizing "ppack(t_1,t_2)" has one of the two forms, "OI(1,EI(t_3,t_4))", and "OI(2,t_5)". A result of the form "OI(1,EI(t_3,t_4))" may be read as the term part of a construction

$$OI(1,EI(t3{:}legal(t_1,t_2,t_4),t_4){:}\exists\Lambda(legal(t_1,t_2,\Lambda))){:}\exists\Lambda(legal(t_1,t_2,\Lambda))\vee\neg(\exists\Lambda(legal(t_1,t_2,\Lambda)))$$

This result indicates that a legal assignment of blocks t_1 into bins t_2 does indeed exist, and that t_4 is such an assignment. If on the other hand the result has the form "OI(2,t_5)", then no legal packing is possible.

4.4.2 Two small experiments

As indicated in the introduction to this chapter, the experiments consist of specializing ppack to handle inputs of a particular size and structure by means of the following steps. (1) The term "ppack(t_1,t_2)" is normalized. Here t_1 and t_2 are open terms having the form of the special inputs to be treated; namely, $\ll i_1, \ldots i_k \gg$, and $\ll n,n \ldots n \gg$, where "i_1", . . . "i_k", and "n" are numeric variables. (2) The normal form of "ppack(t_1,t_2)" is subjected to the "simplex" optimization, which makes use of an additional assumption about the structure of the inputs; in particular, it is assumed that $i_1 \geq i_2 \geq i_3 \ldots \geq i_k$. (3) Pruning is applied. The result of all this is a decision tree algorithm (given by a p-term) for the special task of packing k blocks into some particular number of bins of equal size, under the assumption that the blocks have been given in decreasing order of size.

To begin with, we will describe the results of this process for the simplest case which is not absolutely trivial, namely the case where $t_1 = \ll i1,i2 \gg$, and $t_2 = \ll n,n \gg$. First of all, the result of normalizing ppack($\ll i1,i2 \gg$,$\ll n,n \gg$) is:

p_1 =

```
OE (α7)
 LTED(i1,n)
 (OE (α9)
   LTED(i2,n−i1)
   OI(1,EI(#(α7,α9),≪1,1≫))
   (OE (α11)
     LTED(i2,n)
     OI(1,EI(#(α7,α11),≪1,2≫))
     (OE (α13)
       LTED(i1,n)
       (OE (α15)
         LTED(i2,n)
         OI(1,EI(#(α13,α15),≪2,1≫))
         (OE (α17)
           LTED(i2,n−i1)
           OI(1,EI(#(α13,α17),≪2,2≫))
           OI(2,#(α11,α9,α15,α17))))
       OI(2,#(α11,α9,α13)))))
 (OE (α19)
   LTED(i1,n)
   (OE (α21)
     LTED(i2,n)
     OI(1,EI(#(α19,α21),≪2,1≫))
     (OE (α23)
       LTED(i2,n−i1)
       OI(1,EI(#(α19,α23),≪2,2≫))
       OI(2,#(α7,α21,α23))))
   OI(2,#(α7,α19)))
```

This p-term, if written as an ordinary conditional expression, would read:

c_1 =

```
if i1≤n then
 if i1+i2 ≤n then ≪1,1≫
   else
   if i2≤n then ≪1,2≫
     else
     if i1≤n then
       if i2≤n then ≪2,1≫
         else
         if i1+i2 ≤n then ≪2,2≫ else ≪≫
       else ≪≫
 else
 if i1≤n then
   if i2≤n then ≪2,1≫
     else
     if i1+i2 ≤n then ≪2,2≫ else ≪≫
   else ≪≫
```

The above conditional expression would also result from normalizing the ordinary recursive function definition pack on the symbolic inputs $\ll$i1,i2$\gg$ and $\ll$n,n$\gg$ using the reduction rules mentioned in section 2.8, and in addition the permutation rule:

if (if t_1 then t_2 else t_3) then t_4 else t↑!5

$\Rightarrow$ if t_1 then (if t_2 then t_4 else t↑!5) else (if t_3 then t_4 else t↑!5)

Now, the "simplex optimization" consists of removing "pre-decided" case analyses. Another transformation is applied at this stage, namely the replacement of occurences of assumptions when possible by "proofs" of those assumptions from other available information. This last transformation improves the effectiveness of pruning, since it removes apparent but in fact unnecessary dependencies between the facts involved in the computation. Since all of the decision predicates which appear in bin-packing take the form of inequalities b tween linear terms, the simplex algorithm may be used to perform these transformations.

The "simplex transformations" are instances of the following general replacement transformation on proofs. First of all, we define the *set of active assumptions at a node of a proof tree* to be the set of assumptions discharged along the path from the node in question to the root node of the proof. More formally, a formula A is active at a node N if N lies in [using q-term notation]: (1) Π_2 of OE(Π_1:A$\vee$F,Π_2,Π_3), (2) Π_3 of OE(Π_1:F$\vee$A,Π_2,Π_3) (3) Π of II(A,Π), (4) Π_2 of EE(Π_1,$\exists$xA,Π_2). A *replacement* transformation is a transformation which replaces a subproof Π':A rooted at node N of a proof Π by another proof Π'':A of the same formula A, subject to the condition that the open assumptions of Π'' are among those active at N.

The simplex transformations are replacement transformations of a special kind. Consider a subterm of the form "LTED(t_1,t_2)" which appears in a bin-packing p-term. Suppose that one of $t_1 \leq t_2$ or $t_2 < t_1$ follows from the active assumptions at the node at which LTED(t_1,t_2) appears (all of the active assumptions will themselves be linear inequalities). That is to say, suppose that the outcome of executing LTED(t_1,t_2) is pre-decided by the linear inequalities which have already been assumed at the current node in the decision tree. Then the invocation of the lemma LTED can be removed in favor of a small proof of "$t_1 \leq t_2 \vee t_2 < t_1$" by means of an $\vee$-introduction from one or the other of the results "$t_1 \leq t_2$", or "$t_2 < t_1$". Specifically, that proof will have the form

$$OI(k,AX([F_1,F_2,\dots F_n \supset F_0])(AS(F_1),AS(F_2)\dots AS(F_n)))$$

where k is either 1 or 2, where F_0 is either $t_1 \leq t_2$ or $t_2 < t_1$, and where $F_1,F_2, \dots F_n$ are the various inequalities which are active assumptions at this point in the decision tree, and which are needed to conclude that F_0 holds. This is exactly the replacement which is performed by the first simplex transformation - except that the replacement is carried out in the language of untyped p-terms; thus the replacing term has the form $OI(k,\#(\alpha_1, \dots ,\alpha_n))$, where the α_i are proof variables.

Now, let us consider *the second transformation - the dependency removal transformation*. Suppose that an assumption $AS(F_0)$ in a specialized bin-packing proof follows from other assumptions which are active at the node where the assumption appears. Then the various results which are derived using the assumption have the appearance of depending on that assumption, but the dependency is in a sense unreal - it could be dispensed with. If we wish to make the best use of pruning, then apparent dependencies of this kind should be eliminated. So we use the simplex method to replace assumptions $AS(F_0)$ by proofs of those assumptions from other assumptions which are currently in effect. The form of the proofs with which assumptions are replaced is

$$AX([F_1,F_2,\dots F_n \supset F_0])(AS(F_1),AS(F_2)\dots AS(F_n))$$

where $F_1,F_2, \dots F_n$ are the formulas needed to establish F_0. In p-term notation, this has the form $\#(\alpha_1, \dots ,\alpha_n)$. Note that the formulas which are associated with proof variables in the bin-packing p-terms can be determined by finding the OE operator which binds the variable, and looking at its first argument "LTED(t_1,t_2)"; if the variable in question appears in the second premise to this OE operator then the associated formula is "$t_1 \leq t_2$", and otherwise it is "$t_2 < t_1$".

One more piece of information remains to be specified about the simplex transformations. It may happen that several distinct proofs can be used to replace a single assumption or "LTED" invocation; the inequality in question might follow from several different subsets of the the currently active set of assumed inequalities. We have not said how a choice among several such possibilities is to be made. In fact only one possibility, namely the one generated by the following algorithm, is considered. Let $F_1,F_2, \dots F_n$ be a list of all the assumptions active at a given node in the order of "innermost" to "outermost"; that is, F_1 is the assumption discharged nearest the current node, while F_n is the assumption discharged nearest the root of the proof. In attempting to find a minimal subset of $\{F_1, \dots F_n\}$ from which a formula F_0 can be derived, our algorithm proceeds in the following way. First it checks (using the simplex method) whether F_0 follows from $\{F_1\}$. If not then it checks $\{F_1,F_2\}$, $\{F_1,F_2,F_3\}$, and so on, until it finds a least j such that F_0 follows from $\{F_1, \ldots .F_j\}$, or until it is determined that F_0 does not follow from the entire set $\{F_1, \ldots F_n\}$. In the latter case, we are done, and return a negative answer. In the former case, we scan through the set again, in the order $F_1, \ldots .F_n$ in which it is given. For each element F_i considered, an attempt is made to remove F_i from the set; the attempt is deemed successful if the reduced set still implies F_C After removing or attempting to remove each F_i in turn, we evidently have a mininimal set of inequalities with the desired property. It is this set which is returned by the algorithm. This algorithm was the first that came to mind, and, because it produced good results, we did not try another.

In each of the simplex transformations, the inequalities $i_1 \geq i_2, i_2 \geq i_3, \ldots .i_{k-1} \geq i_k$, are assumed as "background" information. That is to say, whenever we used the phrase "F_0 follows from $\{F_1 \ldots F_n\}$" in the above, we meant "F_0 follows from $\{F_1 \ldots F_n\}$ and $\{i_1 \geq i_2, i_2 \geq i_3, \ldots .i_{k-1} \geq i_k\}$".

Note that the only property of the bin-packing proofs of which simplex transformations make special use is the fact that the decision predicates have the form of linear inequalities. Transformations of the same kind - namely, replacements of case analyses and replacement of assumptions - can be applied to any proof under the condition that a decision procedure is available for the case predicates which appear in the proof. Thus, the special purpose part of the simplex transformations is just the simplex algorithm itself.

As mentioned earlier, the simplex algorithm used in the implementation was not written by the current author. Rather, a "canned" simplex package, written (in MacLISP) by Greg Nelson, was imported into the proof manipulation system.

The result of applying the simplex transformations to the normal form of "pack(≪i1,i2≫,≪n,n≫)" given earlier, and then normalizing again is:

p_2 =

```
    OE(α25)
      LTED(i1,n)
      (OE(α27)
        LTED(i2,n−i1)
        OI(1,EI(#(α27),≪1,1≫))
        OI(1,EI(#(α25),≪1,2≫)))
      OI(? #(α25))
```

Written as an ordinary conditional expression, this is:

c_2 = if i1$\leq$n then (if i1+i2$\leq$n then ≪1,1≫ else ≪1,2≫) else FAIL

Note that the first of the two simplex optimizations - namely, the one which removes pre-decided case analyses - could as easily have been applied the conditional term c_1, and the result would have been c_2. Thus, so far, no use has been made of the additional dependency information which the p-term contains, but which the conditional term does not. However, pruning is applicable to p_2, yielding:

p_3 = OE(α29) LTED(i1,n) OI(1,EI(#(α29),≪1,2≫)) OI(2,#(α29))

Written as a conditional expression, this is:

c_3 = if i1$\leq$n then ≪1,2≫ else FAIL

Thus p_3 tries only one packing, namely ≪1,2≫. If any packing works, then this one must. This fact is "automatically realized" by the dependency analysis involved in pruning.

Note that p_3 computes a different function from that computed by p_2. Also note that p_2 is the optimal (ie smallest and fastest) conditional expression for *computing the function* λi1 i2 n. pack(≪i1,i2≫,≪n,n≫) with i1$\geq$i2. Thus, it is only by using a transformation (such as pruning) which modifies the extensional meaning of computational descriptions that we are able to achieve the improvement which p_3 represents over p_2.

As mentioned in the introduction to this chapter, it is not feasible to perform stages (1) and (2) of the specialization separately for the larger examples. The reason for this is that the p-terms which result from stage (1) alone are too large to fit in memory in the current implementation. Thus the simplex transformations and normalization were run in parallel; the normalizer was modified so as to apply the case analysis removal procedure when called upon to "normalize" an expression of the form "LTED(t_1,t_2)". Assumption replacement was implemented in a similar manner.

We now present the results of another small experiment, namely, the experiment in which pack(≪i1,i2,i3≫,≪n,n≫) is specialized. First of all, the worst case running time of the original version of pack (or equivalently of ppack with our call-by-value normalizer) with $i1 \geq i2 \geq i3$ is 10, where running time is measured in number of comparisons. More precisely, there are numerals a,b,c,d with $a \geq b \geq c$ such that the number of comparisons made in the course of the execution of pack(≪a,b,c≫,≪d,d≫) by a standard call-by-value evaluator for conditional expressions is 10, and furthermore this is the largest number of comparisons which will be made in any execution of pack applied to an input with this form. The worst case running times for pack reported here and below were computed using a program which searches through all possible execution paths (ie sequences of comparisons) of pack when applied to an input of the special form under consideration; the length of the longest such path is returned. The simplex algorithm is used to determine which execution paths are possible, and which are not.

The result of normalizing and applying the simplex transformations to pack(≪i1,i2,i3≫,≪n,n≫) is:

```
OE (α11)
  LTED(i1,n)
  (OE (α13)
    LTED(i2,n−i1)
    (OE (α15)
      LTED(i3,n−i1−i2)
      OI(1,EI(#(α15),≪1,1,1≫))
      OI(1,EI(#(α13),≪1,1,2≫)))
    (OE (α17)
      LTED(i3,n−i1)
      OI(1,EI(#(α11,α17),≪1,2,1≫))
      (OE (α19)
        LTED(i3,n−i2)
        OI(1,EI(#(α11,α19),≪1,2,2≫))
        OI(2,#(α19)))))
  OI(2,#(α11))
```

Pruning when applied to the above p-term yields

```
OE (α21)
  LTED(i1,n)
  (OE (α29)
    LTED(i3,n−i2)
    OI(1,EI(#(α21,α29),≪1,2,2≫))
    OI(2,#(α29)))
  OI(2,#(α21))
```

Written as an ordinary conditional expression, this is:

```
if i1 ≤ n then
  if i2+i3 ≤ n then ≪1,2,2≫ else FAIL
  else FAIL
```

Note that pruning again yields an optimal algorithm for the special case considered - an algorithm which computes a different funticon from that originally computed by pack.

4.4.3 An algorithm for packing six blocks into three bins

The results of the experiment concerning the packing of six blocks into three bins were described in general terms in the introduction to this chapter. The end product of that experiment - that is to say, the algorithm produced at the last stage of the three stages of optimization - is given below as an ordinary conditional expression.

```
if  i1 ≤ n then
    if  i2+i3 ≤ n then
        if  i1+i6 ≤ n then <1,2,2,3,3,1>
            else
            if  i4+i5+i6 ≤ n then <1,2,2,3,3,3>
                else FAIL
        else
        if  i2+i4 ≤ n then
            if  i1+i6 ≤ n then <1,2,3,2,3,1>
                else
                if  i3+i5+i6 ≤ n then <1,2,3,2,3,3>
                    else FAIL
            else
            if  i3+i4 ≤ n then
                if  i2+i5 ≤ n then
                    if  i1+i6 ≤ n then <1,2,3,3,2,1>
                        else
                        if  i2+i5+i6 ≤ n then <1,2,3,3,2,2>
                            else
                            if  i3+i4+i6 ≤ n then <1,2,3,3,2,3>
                                else FAIL
                    else
                    if  i3+i4+i5 ≤ n then
                        if  i6+i2 ≤ n then <1,2,3,3,3,2>
                            else
                            if  i3+i4+i5+i6 ≤ n then <1,2,3,3,3,3,>
                                else FAIL
                        else FAIL
                else FAIL
    else FAIL
```

4.4.4 Table of other results

The following table summarizes the results of the remaining experiments. Six numbers are associated with each experiment. These quantities are:

(1) P. This is the worst case running time of pack applied to inputs of the form under consideration.

(2) EP. The performance of the general purpose algorithm pack in treating special cases where the bins are all of the same size is very bad. One reason for this is that no use is made of symmetries introduced by the equal sizes of the bins; each of various packings which are equivalent under renaming of bins is considered separately. It was of interest to compare the performace of our optimized special purpose algorithms with the performance of an algorithm with the same design as pack, but which takes the symmetries introduced by equal bin sizes into account. That algorithm is as follows:

```
epack(X,s,k) ← epack1(X,<>,1,s,k)

epack1(X,B,n,s,k) ←

        if n≤lnth(B) then
           if X:1≤B:n then
              {λ z.
                  (if z≠FAIL then n @ z
                        else epack1(X,B,n+1,s,k))}
              (epack1(tl(X),set(B,n,B:(n − X:1)),1,s,k)
              else epack1(X,B,n+1,s,k)
           else
           if k≥1 ∧ (X:1≤s) then
              {λ z. if z≠FAIL then (lnth(B)+1) @ z
                          else FAIL} (epack1(tl(X),B*<X:1>,1,s,k-1))
              else FAIL
```

The algorithm epack(X,s,k) searches for a packing of the blocks in the list X into k bins each of size s. The subprogram epack1(X,B,n,s,k) searches for a packing of the blocks X into a collection of bins described by the inputs B,s, and k. The initial elements of this collection are just the bins whose sizes are given in B, while the remainder of the collection consists of k bins each of size s. As in packb, the first block X:1 must be placed in a bin whose index is at least n. The behavior of epack1 resembles that of packb, except that it keeps track of which bins are still empty. A block is placed in an empty bin only if the attempt to place it in a non-empty bin leads to failure. In contrast to packb, epack1 attempts at most one placement of any block into an empty bin. The term "B*≪X:1≫" in epack1 denotes the result of appending the list "≪X:1≫" onto the end of the list B.

The number EP represents the worst case running time of epack.

Note that, even if it had turned out that the "hand-optimized" algorithm was more efficient than the specialized algorithms which we produce by automatic methods, it would not follow that the automatic methods are not of use. An automatic specialization method such as the one currently under discussion starts with a general algorithm and with a description of the special form of the inputs to be considered; the output of the method is then a specialized algorithm which deals with inputs of that special form. The most direct measure of the effectiveness of the specialization method is given by a comparison of the output of the method with the original algorithm, and not with some third algorithm (such as epack) *produced by a person to handle inputs of the special form.* A separate matter of interest is to compare human and automatic performance in this regard as we are doing at the moment. As it happens, and as will be seen, our automatically specialized algorithms are in fact faster than the algorithm epack given above.

(3) D is the depth of the decision tree produced by applying normalization and the simplex transformations to pack(≪i1, . . . in≫,≪n,n, . . .n≫). Equivalently, D is the number of comparisons made along the longest path down the decision tree; that is to say, the "running time" of the decision tree.

(4) D_p is the depth of the decision tree produced by applying pruning to the tree of (3) immediately above.

(5) S is the size of the decision tree of (3) measured as the number of decision points; equivalently, S is the number of occurences of "LTED" in the p-term.

(6) S_p is the size of the pruned decision tree of (4).

In the table, the above quantities are arrayed in the form:

P
EP
D $\quad$ S
D_P $\quad$ S_P

The effectiveness of pruning is indicated by the differences between D and D_P, and between S and S_P. The table of results is as follows. Occurences of "*" in the table indicate that the relevant decision trees could not be constructed because of lack of memory space.

BLOCKS	2	3	4	5
3	10 4 4 5 2 2			
4	14 6 6 10 4 4	48 7 7 13 2 2		
5	38 17 7 27 6 12	66 10 10 33 4 4	260 11 11 37 2 2	
6	58 27 17 62 12 40	174 27 14 87 8 15	356 15 * * 4 4	1630 16 * * 2 2
	BINS			

4.5 Summary

The results of the experiment show that prunable redundancies can indeed arise in the specialization of a simple combinatorial algorithm, and consequently that pruning can be of use in specialization. It is of course possible that equally good specialized algorithms for the particular problem treated - namely, bin-packing - could have been arrived at by a head on attack. For example, one such attack would involve manipulating the propositional formulas which result from unwinding the definition of a legal packing as applied to inputs of restricted size. However, as has been remarked earlier, the methods by which the specialization was done are for the most part completely general in their applicability; the only special property of the bin-packing problem which was used was the decidability of linear inequalities. The machinery of normalization, and pruning, and proof replacement may be applied to any proof whatever. The experiments should be seen as a first test of the utility of this general machinery. Our purpose was not to develop fast special purpose bin-packing algorithms, but to investigate pruning in a setting where its effects could be easily isolated.

Chapter 5

Other Applications

Until now, we have restricted attention to the use of proof manipulation in specializing algorithms. The purpose of this chapter is to briefly indicate other computational applications of the proof manipulation technology which has been described in the course of this thesis, and at the same time to outline some connections between our work and other traditions of work within computer science. Applications to two kinds of computational problems other than specialization will be considered, namely, applications to the automatic construction of proofs (from proof fragments; section 5.1), and to the analysis of change (section 5.2).

5.1 Automatic construction of proofs

As emphasized in the introduction to this thesis, most work in computer science to do with formal proof systems has concerned the automatic construction of proofs, and not their manipulation. Generally speaking, the aim of such work has been to provide automatic means for determining the truth values of propositions; a proof of a proposition is constructed in order to determine that it is valid. Automatic proof construction (or "automatic deduction") in its most pure and ambitious form involves starting with an arbitrary formula of an expressive language (eg the predicate calculus) as the only input data; the output is either an indication that a proof has been found, or an indication of failure. Other forms of automatic deduction make use of additional input data beyond the formula to be proved; for example sets of of "rules" for backward chaining [Shortliffe 1974], or sets of programs which indicate in explicit algorithmic terms how certain problems are to be reduced to subproblems [Hewitt 1971]. It is traditional within artificial intelligence to refer to this additional input data as "knowledge".

Normalization constitutes, in a certain sense, a method for automatically constructing proofs; a normal proof of a proposition is automatically constructed from an arbitrary proof of that same proposition. In this case, the "additional input data" in the sense of the last paragraph consists of the original proof. From the point of view of automatic deduction, normalization is of no use, since the additional input data with which it starts is already satisfactory evidence for the truth of the proposition in question. However, by liberalizing the requirements which apply to proof procedures and lemmas (section 2.5) it is possible to use the machinery of normalization as developed in chapters 2 and 3 for constructing a normal and complete proof of a proposition starting from an *incomplete* proof of the same proposition. Under these conditions, the normalization of the incomplete proof includes a search for evidence for propositions, and thus constitutes a form of automatic deduction in the traditional sense.

Specifically, let us drop the following requirements concerning lemmas: (1) the requirement that lemmas must be true formulas, (2) the requirement that lemmas may not appear in the proofs constructed by proof procedures, and (3) the requirement that a proof procedure may not return "FAIL" when applied to closed arguments. Also, we will now allow proof procedures to produce proofs which make use of assumptions which are active at the point where a lemma appears. (The notion of an active assumption is defined in section 4.4.2.) We retain the requirement that all *axioms* be true. As a result of the removal of requirements 1 - 3, it is now possible to construct incomplete "proofs" of incorrect formulas - proofs which proceed from false lemmas to false conclusions. However, the main point here is that the process of normalization - exactly as described in chapters 2 and 3 - *may* have the effect of removing appearances of lemmas - thus converting an incomplete proof of a formula whose truth is in doubt into a complete and reliable proof of that same formula.

If normalization is implemented in a call-by-value manner as described in section 4.1, then the normalization of an incomplete proof corresponds in a direct way to proof search by backward-chaining through implications - in other words to "subgoaling". Specifically, in the course of normalizing a proof $\Pi:\varphi$ containing lemmas $L_1:\forall \underline{x}\Psi_1(\underline{x})$, $L_2:\forall \underline{x}\Psi_2(\underline{x})$ . . . $L_n:\forall \underline{x}\Psi_n(\underline{x})$, the proof procedures for some or all of the lemmas are invoked. (The invocation of a proof procedure corresponds roughly to an attempt to "match" a subgoal.) When the proof procedure for, say, L_i, is called with input $\underline{t}$, the procedure will either fail (corresponding the failure of a subgoal in backward chaining), or return a proof Π_i of $\varphi(\underline{t})$. In the latter case, Π_i is then normalized. Since Π_i may itself contain lemmas, the normalization of Π_i will in general involve further backchaining. If the end result of normalization is a proof in which no lemmas any longer appear, then the endformula φ has been "proved"; this corresponds to a successful search for a proof by backward chaining. (In particular, this corresponds to backward-chaining without backtracking; however, the addition of backtracking to the mechanism of normalization is a straight-forward matter.)

Let Π be an incomplete proof of a universal formula $\forall x\varphi(x)$. Then it will often happen that the normalization of Π fails to yield a complete proof of $\forall x\varphi(x)$, but at the same time, normalization of the proof

$$\begin{array}{c}\Pi\\ \forall\text{E}\dfrac{\forall x\varphi(x)}{\varphi(t)}\end{array}$$

for a particular term t does yield a complete proof. This can come about in the following way. The normalization of $\Pi(t)$ will in general select a smaller and more specialized set of "subgoals" (that is, lemmas for which proof procedures are invoked) than the normalization of Π; in theorem proving language, the normalization of Π determines the particular set of subgoals needed to verify each instance $\varphi(t)$ of the general formula $\forall x\varphi(x)$ - different sets of subgoals will be generated for different instances. The subgoals generated by normalizing $\Pi(t)$ may be satisfyable even though those generated by normalizing Π are not. In this case, Π does not provide evidence for the truth of the general statement $\forall x\varphi(x)$ (indeed, $\forall x\varphi(x)$ may not be true), but does indicate a method for attempting to construct evidence for instances $\varphi(t)$ of the general statement.

In the case where φ is existential, that is, where $\varphi(x)=\exists y\psi(x,y)$ for some ψ, a successful normalization of

$$\forall E\dfrac{\begin{array}{c}\Pi\\ \forall x\exists y\psi(x,y)\end{array}}{\exists y\psi(x,t)}$$

yields a value for y; thus Π describes an algorithm for computing a *partial* function satisfying the specification ψ. The computation in question involves a mixture of ordinary computation (normalization), and proof search by backward chaining. In this respect, normalization of partial proofs resembles the behavior of "pattern matching languages" such as Planner[Hewitt 1971] and its successors, where ordinary computation is mingled with subgoaling. More will be said about this resemblance later.

The correspondence between normalization and familiar kinds of backward chaining is enhanced if the proof procedures for lemmas proceed by searching for a "match" between the lemma to be proved and the endformulas of proofs in a pre-existing data base. For example, suppose that one starts with a data base $\{\Pi_1, \ldots \Pi_n\}$ of incomplete proofs of universal formulas. Suppose further that all lemmas which appear in proofs of the data base are $\forall\exists$ formulas. Finally, suppose that the following uniform proof procedure is supplied for all lemmas: the procedure, when given input $\underline{t}$ for a lemma $L{:}\forall\underline{x}\exists y\psi(\underline{x},y)$, scans the data base $\{\Pi_1, \ldots. \Pi_n\}$ looking for a proof $\Pi_i{:}\forall\underline{z}\varphi(\underline{z})$ such that the formulas $\psi(\underline{t},y)$ and $\varphi(\underline{z})$ *unify* in the sense that there is vector of terms $r_0,r_1,r_2 \ldots r_k$ with $\psi(\underline{t},r_0) = \varphi(r_1,r_2 \ldots r_k)$. If such a proof Π_i is found, then the procedure returns the proof

$$\exists I\dfrac{\forall E\dfrac{\begin{array}{c}\Pi_i\\ \forall\underline{z}\varphi(\underline{z})\end{array}}{\varphi(r_1,r_2 \ldots r_k)}}{\exists y\psi(\underline{t},y)}$$

If no such proof can be found, the procedure returns "FAIL". The similarities to Planner and its successors, and also to the "logic programming language" PROLOG [Kowalski 1974], should now be evident. In particular, the behaviour of both MicroPlanner, and of PROLOG programs, can be closely matched by the machinery just described. The proofs $\{\Pi_i\}$ correspond to consequent theorems of MicroPlanner, and to the horn clauses of PROLOG. The value returned by a successful execution of a PROLOG program corresponds to the realization which may be extracted from a normal proof of an existential theorem.

As has been convincingly demonstrated by work with PROLOG, a person who knows in general terms how backward-chaining works is in practice able to express an arbitrary algorithm as a set of implicational formulas; the execution of the algorithm takes place when a backward-chaining theorem prover (eg, the PROLOG interpreter) is given those formulas as axioms, and a goal which encodes the input to the computation. (One also needs a mechanism for extracting an output value from a proof; in PROLOG, this output is constructed in the course of the search for the proof.) It is of course essential that sets of implications be constructed with an algorithm explicitly in mind; a set of implicational formulas which are chosen solely according to the criteria of Tarskian truth and completeness are exceedingly unlikely to be of any computational use, regardless of the theorem prover used. (This is analogous to the observation that a proof of an $\forall\exists$ theorem which is constructed soley according to "mathematical" criteria such as validity and elegance is unlikely to be of much computational use when executed by normalization.) Kowalski[1974] has discussed the advantages of describing algorithms by sets of formulas and executing them by use of a backward-chaining theorem prover. As we have shown, it is possible to mix normalization with backward chaining; presumably, this should allow the benefits of the two forms of computation to be realized simultaneously.

We remark on two additional aspects of automatic proof construction using normalization:

(1) Note that what Stalhman and Sussman[1977] have called dependency directed backtracking "comes for free" in normalization with pruning. Suppose that one wishes to normalize a proof

$$\vee E \; \dfrac{\begin{matrix} & [A] & [B] \\ \Pi_1 & \Pi_2 & \Pi_3 \\ A\vee B & C & C \end{matrix}}{C}$$

whose main inference is $\vee$-elimination. Suppose further that C is a Harrop formula, and that normalization of Π_1 does not decide between A and B. (Evidently, the requirement that only non-Harrop axioms appear in proofs may be dropped in the case where the endformula is a Harrop formula; consequently the normal form of Π_1 may fail to decide between A and B - for example, Π_1 might consist simply of the axiom, "$A \vee B$".) Then, in the usual case, it will be necessary to normalize both Π_2 and Π_3. However, if the normal form Π_2' of Π_2 does not make use of the assumption A, pruning allows us to produce Π_2' as the end result, and thereby to dispense with the treatment of Π_3. Thus dependency information can be used to reduce the amount of search or "backtracking", just as it does in the various systems for "dependency based reasoning" which have been developed by workers in Artificial Intelligence (see London[1978], Doyle[1978], Shrobe[1979]). Also note that in normalization, dependency directed backtracking does not rely on "non-hierarchical contexts" or non-monotonic inferences.

(2) A complete proof of a formula $\forall x \exists y \varphi(x,y)$ provides evidence for the truth of $\forall x \exists y \varphi(x,y)$, and in addition describes a method for computing a function f with $\forall x \varphi(x,f(x))$. As a consequence, the normalization of an incomplete proof $\Pi{:}\forall x \exists y \varphi(x,y)$ consitutes both a search for evidence, and a search for an algorithm with certain properties; in the terminology of computer science, normalization can serve as a method for the synthesis of complete programs from program fragments. (For comparison with program synthesis for PROLOG see [Clark and Sickel, 1977]). Notes: (a) If normalization is implemented as a semi-automatic procedure - a procedure in which a human user has the option of interactively constructing proofs of lemmas - then we arrive at a "refinement" method for constructing programs very much like that developed by Bates[1979]. (b) A single proof transformation, namely pruning, can have the effect of improving the efficiency of a computation at "run-time" (as explained in the last paragragh), or of optimizing an algorithm, depending on whether the transformation is applied in the course of computing a value, or to a proof of an $\forall\exists$ formula.

It is also worth considering the case where normalization of $\forall x \exists y \varphi(x,y)$ produces a proof Π' which is not complete. Here, Π' may still be used to compute values of y with $\varphi(x,y)$ from values of x in the manner described earlier; the computation will not consist of "pure" normalization, but will involve backward-chaining through lemmas as well. What, then, is the significance of the passage from Π to Π'? In the scheme for the execution of incomplete proofs with which we are currently concerned, the burden of computation is shared between automatic deduction (perhaps in the form of "matching"), and pure normalization. When $\Pi{:}\forall x \exists y \varphi(x,y)$ is executed, all computation (including both pure normalization and automatic deduction) which is possible in the abscence of a concrete value for x is carried out. When a concrete value for x is supplied, the remainder of the computation is performed. Thus the passage from Π to Π' constitutes a kind of optimization; all work which can be done without knowing the value of x is carried out first, and, as a consequence, this work does not have to be repeated each time Π' is run.

5.2 Analysis of change

Consider a situation in which one is obliged to solve a series of problems $P_1, P_2, \ldots P_n$, where P_{i+1} is only "slightly different" from P_i. Then it may happen that the same solution works for many consecutive problems. It is useful in this situation to determine conditions under which a small change in a problem leaves the correctness of a solution intact; if the difficulty of evaluating such conditions is small compared to the effort involved in constructing a new solution, then the total effort needed for solving $P_1, P_2, \ldots P_n$ can be reduced.

In Artificial Intelligence, the task of determing the effects of small changes is referred to as the "frame problem" [McCarthy 1969]. The use of proofs as descriptions of algorithms can provide aid in attacking the frame problem, in the following way. Suppose that when a problem P is solved, one constructs not only a solution S, but also a proof Π that S really is a solution of P. Then Π provides an explicit description of the features of the problem upon which the success of S depends. If P is changed slightly, one is able to see, by inspecting the proof Π, whether any feature relevant to the success of S has been modified. Now, if one uses a proof to describe a method for solving a problem, then the execution (ie normalization) of the proof when applied to a particular problem yields not only a solution, but also a specialized proof that the solution is correct; and, as we have said, this proof can be used in the analysis of change.

This idea is illustrated by the following schematic example. Consider the problem of of computing an output value v with $\varphi(\underline{t},v)$ when given a vector $\underline{t} = t_1, \ldots t_n$ of inputs. Suppose that an algorithm for doing the computation is given by a proof Π of $\forall\underline{x}\exists y\varphi(\underline{x},y)$ and that the result of executing $\Pi(\underline{t})$ is a proof Π' of $\exists y\varphi(\underline{t},y)$ which provides v as a value for y. In the general case, Π' will make use of properties of some but not all of the inputs $t_1, \ldots t_n$. Suppose then that a "slightly different problem" is presented - namely the problem of computing v' with $\varphi(\underline{t}',v')$, where the vector $\underline{t}'$ differs from $\underline{t}$ in only a few entries. If the entries in which $\underline{t}'$ differs from $\underline{t}$ do not include any of the entries whose properties are mentioned by Π', then $\varphi(\underline{t}',v)$ holds, and the computation does not need to be repeated.

The same kind of analysis of change can be carried out without using proofs. Suppose that, in the above schematic example, the computation of v from $\underline{t}$ is carried out by the execution of an ordinary program $p(x_1, \ldots x_n)$ rather than by the normalization of a proof. Then a trace of the execution of $p(t_1, \ldots t_n)$ will indicate which among the values $t_1, \ldots t_n$ have been used in the computation and which have not, thus providing the same kind of dependency data as is supplied by the normal proof $\Pi':\exists y\varphi(\underline{t},y)$. However, the normal proof Π' in general provides a more thorough and more useful analysis of dependencies that the corresponding program trace. To see how this can come about, compare the execution of a conditional expression

$$\text{if } r_1 \text{ then } r_2 \text{ else } r_3$$

with the normalization of the corresponding proof:

$$\vee E\;\dfrac{\begin{array}{ccc} & [A] & [B] \\ \Pi_1 & \Pi_2 & \Pi_3 \\ A \vee B & C & C \end{array}}{C}$$

Suppose that (1) r_1 evaluates to "TRUE", (2) the normal form of Π_2 does not contain the assumption A, and (3) an input t_i appears in r_1 (and in Π_1) but not in r_2 (nor in Π_2). Then, in a trace of the execution of "if r_1 then r_2 else r_3", the outcome will appear to depend on t_i, but the corresponding normal proof will reveal that the *correctness* of the outcome is independent of t_i.

Thus, in the analysis of change as in the specialization of algorithms, proofs provide additional data about the dependencies between facts involved in a computation, and this additional data can be exploited to avoid redundant computation.

Analysis of change of the kind which we have been discussing - based however on the use of programs, and not proofs, as descriptions of algorithms - has been used in a number of settings within computer science. To take a simple example, the conventional program optimization which is known as *code motion* [Aho and Ullman 1973] involves analysis of change in the context of iterative computation. In the typical kind of code motion, an assignment statement "$v \leftarrow t$" is moved out of an inner loop when it is determined that the variables appearing in t do not change in the loop. By using a proof for describing the computation of the value to be assigned to v, this analysis of change might be improved - specifically by determination of the conditions under which the *correctness* of the value computed depends on variables which change in the loop. A related idea is worked out in a paper of Katz[1978] concerning the use of proofs of invariant assertions in optimizing iterative descriptions of computation.

Other examples are provided by *constraint systems* such as those developed by Stallman and Sussman[1977], and Borning [1979], and by "dependency based reasoning systems" such as those of Shrobe[1979], London[1978] and Doyle[1978]. In these systems, situations - such as the state of an electrical circuit [Stallman and Sussman 1977] - are represented in such a way that the dependencies among the facts and values which describe the situation are explicitly recorded. When the situation changes, or when an assumption about the situation is added or withdrawn in the course of automatic deduction, the dependency information is used to determine what aspects of the situation have been affected, and what computation has to be done to update the representation. For the reasons given above, the use of proofs as descriptions of algorithms may be expected to improve the analysis of dependencies upon which these systems rely.

Appendix A

Comparison to Extraction Methods from Proof Theory

Traditional proof theory provides two kinds of methods for the execution of proofs. First there are the methods which operate by transformation of the proofs themselves. The normalization procedure of Prawitz[1965] described in chapter 2 belongs to this class, as does the cut-elimination procedure of Gentzen [1969] for the calculus of sequents. Second, there are methods which involve extracting "programs" of one kind or another from proofs; it is then the program which is executed, and not the proof itself. Examples of methods of the latter kind are the recursive realizability interpretation of Kleene[1945], the Dialectica interpretation of Gödel[1958], and the modified realizability interpretation of Kreisel[1959] for analysis.

The normalization method, and the modifications to it which we have made in order to increase efficiency, have of course been discussed at length in this thesis. The purpose of this appendix is to compare the methods which we use to the other family of execution methods from proof theory - namely, the functional and realizability interpretations. The account which follows is intended for the reader who is familiar with these interpretations.

In general terms, the situation is this. The "*programs*" extracted by the three interpretations mentioned above are Gödel numbers of partial recursive functions in the case of recursive realizability, and typed λ-calculus terms in the other two cases. As shown by Mints[1977], the various programs extracted by these interpretations from a proof of $\forall x \exists y \varphi(x,y)$ all compute the same function as does normalization. Furthermore, the convertability results of Mints, and the commutativity results of Diller [1979] show that it is not only the function computed which remains fixed under these interpretations, but also the form of the computation sequences which arise when the function is applied to a particular argument.

The programs extracted by the functional and realizability interpretations mentioned above resemble the untyped p-terms which we extract from proofs in that both the p-terms and the programs contain the information in a proof which is relevant to execution but leave out most of the rest of the data in the proof. The interpretations differ among themselves in the efficiency of the programs which they extract, but, in one case - namely modified realizability - the extracted programs are as consise and computationally efficient as p-terms. The Dialectica interpretaion also produces "good code", but to a somewhat lesser extent. In the case of recursive realizability, efficiency depends on the particular godel numbering and interpreter used.

For our purposes the differences between p-terms and programs are crucial, since p-terms contain the dependency data needed for pruning, whereas the programs do not. In order to specialize algorithms by symbolic execution and pruning as we have done in the bin-packing experiments of chapter 4, we need a form of computational description which meets both of the following requirements: (1) Symbolic execution of the description must be tolerably efficient. (2) The dependency data needed by pruning must be present in the description, and further, this dependency data must be preserved in the course of symbolic execution. Now, normalization as described by Prawitz[1965] meets the second requirement but not the first, whereas, from what we have just said, the programs extracted by the functional and realizability interpretations meet the first requirement but not the second. Thus none of the tools from traditional proof theory is adequate for performing the kinds of manipulations on algorithms which have been the central concern of this thesis, and for this reason it was necessary to use a new form of computational description - the p-term.

For a more explicit formulation of the relationship between p-terms and the programs extracted by the interpretations, we will need the following notation. Let γ_1 be the procedure which extracts untyped p-terms from proofs, and let γ_2 be the extraction procedure for any one of the interpretations. The modified realizability and Dialectica interpretations extract *typed* λ-calculus terms from proofs; however, it is convenient here to regard the terms extracted by γ_2 as terms of the ordinary untyped λ-calculus. This is an inessential modification, since the type information contained by λ-terms is not needed for normalization and cannot help in pruning. With this taken into account, there is a procedure γ_3 for extracting programs *from p-terms* such that the diagram,

$$\begin{tikzcd} {} \arrow[r, "\gamma_1"] \arrow[dr, "\gamma_2"'] & {} \arrow[d, "\gamma_3"] \\ & {} \end{tikzcd}$$

commutes.

Thus, p-terms lie "on the way" from proofs to programs. Furthermore, the map γ_3 is many-to-one; there is no way of getting the p-term back from the program extracted from it.

In part (α) of section A.1, we will describe γ_3 in general terms for the modified realizability interpretation, and show in part (β) that pruning cannot be used in connection with programs produced by this interpretation. The treatment of recursive realizability is essentially identical. In section A.2, the Dialectica interpretation is discussed. The example which shows that pruning does not apply for modified realizability interpretation or for recursive realizability also works for the Dialectica interpretation.

For the current purposes, it is convenient to restrict our attention to a theory which, roughly speaking, represents the intersection of the theories treated by the various interpretations - namely, the formulation of arithmetic given in section 3.6. The language of this theory is just the standard language of arithmetic; the set of available lemmas consists of the induction schema IND_φ. We have not said what axioms are used, and we don't need to, since the choice of axioms makes no difference to what we have to say. (Note that the standard system for intuitionistic arithemetic arises from one such choice of axioms.)

A.1 Modified realizability and recursive realizability

(α) First we describe the map γ_3 which takes a p-term and rewrites it as a modified realization. What γ_3 does is to replace the special operators OI_1, OI_2, OE, EI, EE of the p-calculus by constructs of the ordinary λ-calculus. Specifically, we use the replacements:

$\# \Rightarrow c$ (where c is a constant symbol; a different constant symbol is assigned to each occurence of "#".)

$\mathrm{OI}_1(t) \Rightarrow \langle 0,t\rangle$

$\mathrm{OI}_2(t) \Rightarrow \langle 1,t\rangle$

$\mathrm{OE}(\alpha,t_1,t_2,t_3) \Rightarrow \text{if } \pi_1(t_1) \text{ then } t_2[\alpha\leftarrow\pi_2(t_1)] \text{ else } t_3[\alpha\leftarrow\pi_2(t_1)]$

$\mathrm{EI}(t_1,t_2) \Rightarrow \langle t_1,t_2\rangle$

$\mathrm{EE}(x,\alpha,t_1,t_2) \Rightarrow t_2[x\leftarrow\pi_1(t_1),\alpha\leftarrow\pi_2(t_1)]$

$\mathrm{IND}(x,t) \Rightarrow \{R(\pi_1(t),\ \lambda y\ z.(\{\pi_2(t)(y)\}(\langle 0,z\rangle)))\}\ (x)$

In the above, R is a conventional recursion operator, to which the reduction rules

$R(t_1,t_2)(\mathrm{succ}\ x) \Rightarrow t_2(R(t_1,t_2)(x))(x)$

$R(t_1,t_2)(0) \Rightarrow t_1$

apply.

The conditional operator "if t_1 then t_2 else t_3" is assumed to take the numeral 0 as TRUE, and the numeral 1 as FALSE. The conditional operator can of course be defined by "if t_1 then t_2 else $t_3 \equiv R(t_2, \lambda x\ y.\ t_3)$". Pairing can also be defined in the typed λ-calculus over arithmetic, but since the types of terms are not available in the current context, we take pairing and projection as primitive. (There is no definable operator in the untyped λ-calculus which has the characteristics of a pairing operator, as shown by Barendregt[1972].)

These replacements preserve the behavior of terms under normalization, as shown by the propositions (1) - (4) below.

(1) If t_1 reduces to t_2 by the application of a single reduction rule of the p-calculus, then $\gamma_3(t_2)$ reduces to $\gamma_3(t_2)$ by the application of a single reduction rule of the λ-calculus.

(2) If t is in normal form (for the p-calculus), then $\gamma_3(t)$ is also in normal form (for the λ-calculus).

(3) Let t be a p-term which has been extracted from a proof (of arithmetic). Then t and $\gamma_3(t)$ both have the uniqueness property. By (1), (2) immediately above, we have $|\gamma_3(t)| = \gamma_3(|t|)$, where $|t|$ designates the normal form of t.

(4) Let t be a p-term which has been extracted from a proof in arithmetic *all* of whose axioms are true. Then there is an assignment of types to the variables and constants of $\gamma_3(t)$ such that the resulting typed λ-calculus term realizes the endformula of the proof in the sense of modified realizability.

Thus, from the point of view of execution as opposed to pruning, there is not much difference between the term extracted from a proof for modified realizability, and the p-term which we extract from proofs.

(β) However, γ_3 destroys the dependency information which is needed by pruning. The problem is that in replacing "$\mathrm{OE}(\alpha, t_1, t_2, t_3)$" by "if $\pi_1(t_1)$ then $t_2[\alpha \leftarrow \pi_2(t_1)]$ else $t_3[\alpha \leftarrow \pi_2(t_1)]$", one looses track of the use, if any, which is made in t_2 and t_3 of the assumption represented by α.

In what follows, we demonstrate this point in a formal way by exhibiting a pair of proofs Π and Π' such that (a) $\gamma_3(\gamma_1(\Pi)) = \gamma_3(\gamma_1(\Pi'))$, and (b) pruning can be applied to Π, but not to Π'. Thus the information which distinguishes between proofs which can be pruned and those which cannot is lost by γ_3. *A fortiori*, the data needed to determine the *outcome* of pruning operations is not p esent in the ordinary λ-terms produced by γ_3.

In order to construct Π, Π', we first need proofs Π_1:A∨B, Π_2:A'∨B, Π_3:$\psi(0)$, Π_3':$\psi(0)$, Π_4:$\psi(0)$, Π_5:$\psi(1)$ such that (a) A and A' are distinct, (b) $\gamma_1(\Pi_1)=\gamma_1(\Pi_2)$ (c) the set of open assumptions of Π_3 is {A'}, (d) the set of open assumptions of Π_3' is {A,A'}, (e) $\gamma_1(\Pi_3) = (\gamma_1(\Pi_3')[\alpha \leftarrow \beta])$ where α is the proof variable assigned to the assumption A, and β the proof variable assigned to the assumption A', (f) the set of open assumptions of Π_4 is {B}, (g) the set of open assumptions of Π_5 is {B}. It is not difficult to construct proofs with these properties. For example, we can take $A=\varphi\vee(0=1)$, $A'=\varphi\vee(1=2)$, and (using the q-notation explained in section 4.2),

$$\Pi_1 = \text{"OE}(\Pi_0{:}\varphi\vee B,\ \text{OI}(\text{OI}(\text{AS}(\varphi),0=1),B),\ \text{OI}(\varphi\vee(0=1),\text{AS}(B)))\text{"}.$$

$$\Pi_2 = \text{"OE}(\Pi_0{:}\varphi\vee B,\ \text{OI}(\text{OI}(\text{AS}(\varphi),1=2),B),\ \text{OI}(\varphi\vee(1=2),\text{AS}(B)))\text{"}.$$

where Π_0 is any proof of $\varphi\vee B$. Then, as desired, $\gamma_1(\Pi_1) = \gamma_1(\Pi_2) = \text{OE}(\alpha,\gamma_1(\Pi_0),\text{OI}_1(\text{OI}_1(\alpha)),\text{OI}_2(\alpha))$. By the requirement (e) above, Π_3 and Π_3' must be identical in form except that uses of the assumption A in Π_3' are replaced by uses of the assumption A'in Π_3. This can be achieved by a trick similar to the one used for Π_1,Π_2 above.

Now, we take

$\Pi =$

$$\vee E\;\dfrac{\overset{\Pi_1}{A\vee B} \qquad \exists I\;\dfrac{\vee E\;\dfrac{\overset{\Pi_2}{A'\vee B}\quad \overset{\Pi_3}{\psi(0)}\quad \overset{\Pi_4}{\psi(0)}}{\psi(0)}}{\exists x\psi(x)} \qquad \exists I\;\dfrac{\overset{\Pi_5}{\psi(1)}}{\exists x\psi(x)}}{\exists x\psi(x)}$$

Π' is just the same as Π, except that Π_3' appears in the place of Π_3. The p-term notation for Π is:

$$t \;=\; OE(\alpha,t_1,EI(0,OE(\beta,t_1,t_3,t_4)),EI(1,t_5))$$

where $t_1=\gamma_1(\Pi_1)=\gamma_1(\Pi_2)$, $t_3=\gamma_1(\Pi_3)$, $t_4=\gamma_1(\Pi_4)$, and $t_5=\gamma_1(\Pi_5)$. The p-term notation for Π' is

$$t' \;=\; OE(\alpha,t_1,EI(0,OE(\beta,t_1,t_3',t_4)),EI(1,t_5))$$

where $t_3'=\gamma_1(\Pi_3')$. The only difference between t and t' is that $t_3=t_3'[\alpha\leftarrow\beta]$. As a consequence, t can be pruned to "$EI(0,OE(\beta,t_1,t_3,t_4))$", whereas t' cannot. However, the difference between t and t' is lost in the course of translation from the p-calculus into the λ-calculus; in passing from t_3 to $\gamma_3(t_3)$ and from t_3' to $\gamma_3(t_3')$, α and β are replaced by the same term, namely $\pi_2(\gamma_3(t_1))$. Specifically,

```
γ₃(t) = γ₃(t') =
        if $\pi_1(\gamma_3(t_1))$ then
            $\langle 0$, if $\pi_1(\gamma_3(t_1))$ then $\gamma_3(t_3)[\beta\leftarrow\pi_2(\gamma_3(t_1))]$ else $\gamma_3(t_4)[\beta\leftarrow\pi_2(\gamma_3(t_1))]\rangle$
            else $\langle 1,\gamma_3(t_5)[\alpha\leftarrow\pi_2(\gamma_3(t_1))]\rangle$
```

As desired, this example demonstrates that pruning cannot be applied to the λ-terms extracted by the modified realizability interpretation. The same example works in the same way for the recursive realizability interpretation. The only difference is that "if then else" and the pairing operator are regarded as operators not on λ-terms but on godel numbers of partial recursive functions.

A.2 The Dialectica interpretation

The Dialectica interpretation [Gödel 1958] extracts somewhat more information from a proof than either the recursive realizability interpretation or the modified realizability interpretation; consequently, it requires separate treatment. In part (α), we describe the map γ_3 from p-terms to Dialectica realizations, and in part (β), we show that pruning is not applicable to the terms extracted by the Dialectica interpretation.

(α) For each formula A of arithmetic, the predicate "f Dialectica interprets A" is expressed by a formula $\forall x D_A(f,x)$ in the theory of functionals of finite type over the natural numbers, where D_A is quantifier free. (In the standard treatments of the Dialectica interpretation, the single universal variable x in $\forall x D_A(f,x)$ is replaced by a vector of variables $\underline{x}$. However, it is convenient for our purposes to use a single universal variable which may range over pairs.)

The difference between the modified realizability interpretation and the Dialectica interpretation may be summarized as follows. In the modified realizability interpretation, a functional which realizes a formula $A \supset B$ is required to produce a realization for B whenever it is supplied with a realization of A. In the Dialectica interpretation, a realization for $A \supset B$ must provide not only a way of getting from realizations of A to realizations of B, but also, roughly speaking, a way of getting from refutations of B to refutations of A. Specifically, a Dialectica realization of $A \supset B$ is a pair $\langle X,Y \rangle$ of functionals such that

$$\forall f\, y.\, (D_A(f, Y(\langle f,y \rangle)) \supset D_B(X(f),y))$$

holds. The functional X takes realizations of A onto realizations of B, just as the corresponding functional for the modified realizability interpretation does. The role of the functional Y is this. Suppose that f is proposed as a realization for A but that, in actuality, f does not realize A. Also suppose that a functional y is given such that $D_B(X(f),y)$ does *not* hold. Then y constitutes a refutation of the proposition that X(f) is a realization of B. What Y does is to take the refutation y, and the functional f, and produce a functional $Y(\langle f,y \rangle)$ which constitutes a refutation of the proposition that f is a realization of A.

In the definition given below, it is convenient to write the realization predicate D_A in the form "$\lambda f\ x.\varphi$" , where φ is a quantifier free formula; this allows us to explicitly indicate occurences of the variables f,x which represent the arguments to the predicate. The definition of D_A by induction on the structure of A is as follows.

(1) Base case. For A atomic, $D_A = \lambda f\ x.A$ (where f and x are new variables not appearing in A)

(2) $D_{A\wedge B} \ = \ \lambda f\ x.(D_A(\pi_1(f),\pi_1(x)) \wedge D_B(\pi_2(f),\pi_2(x)))$

(3) $D_{A\vee B} \ = \ \lambda f\ x.((\pi_1(f)=0 \supset D_A(\pi_2(f),\pi_1(x))) \ \wedge \ (\pi_1(f)=0 \supset D_B(\pi_2(f),\pi_2(x))))$

(4) $D_{\exists y.A} \ = \ \lambda f\ x.(D_A(\pi_2(f),x)[y \leftarrow \pi_1(f)])$

(5) $D_{\forall y.A} \ = \ \lambda f\ x.(D_A(f(\pi_1(x)),\pi_2(x))[y \leftarrow \pi_1(x)])$

(6) $D_{A\supset B} \ = \ \lambda f\ x.\ (D_A(\pi_1(x),\pi_2(f)(x)) \supset D_B(\pi_1(f)(\pi_1(x)),\pi_2(x))$

The map γ_3 for the Dialectica interpretation yields not one but several λ-terms when applied to a p-term t. Namely, it produces (1) a realization X, anu, (2) a term Y_α for each proof variable α which appears free in t. The term X is a Dialectica realization for the endformula of the proof Π from which t was extracted, while for each α, the term Y_α computes refutations (in the sense described earlier) of the open assumption of Π which corresponds to the proof variable α. More precisely, if $\alpha_1, \ldots \alpha_n$ are the proof variables for assumptions $A_1 \ldots A_n$ in a proof Π with endformula C, then the formula

$$\forall \alpha_1, \ldots \alpha_n\ x.$$
$$(D_{A_1}(\alpha_1,Y_{\alpha_1}(x)) \ \wedge \ D_{A_2}(\alpha_2,Y_{\alpha_2}(x)) \ \ldots \ \wedge \ D_{A_n}(\alpha_n,Y_{\alpha_n}(x)) \supset D_C(X,x))$$

holds in the theory of functionals of finite type for some assignment of types to the variables and constants of the Y_{α_i} and of X. We will designate the realization X which is extracted from a term t by "X(t)", and the refutation maps Y_α by "$Y_\alpha(t)$". Some of the clauses of the inductive definition of the extraction map γ_3 are as follows. The remaining clauses have a similar character; the interested reader should have no trouble working them out for himself.

(1) Base case

X: $\alpha \Rightarrow \alpha$;

Y_α: $\alpha \Rightarrow \lambda x.x$

X: # $\Rightarrow$ c [where c is a new constant symbol]

(2) X: $OI(1,t) \Rightarrow \langle 0,X(t)\rangle$

X: $OI(2,t) \Rightarrow \langle 1,X(t)\rangle$

Y_α: $OI(1,t) \Rightarrow \lambda x.\{(Y_\alpha(t))(\pi_1(x))\}$

Y_α: $OI(2,t) \Rightarrow \lambda x.\{(Y_\alpha(t))(\pi_2(x))\}$

(3) $X{:}OE(\alpha,t_1,t_2,t_3)) \Rightarrow$ if $\pi_1(X(t_1))$ then $X(t_2)[\alpha\leftarrow\pi_2(X(t_1))]$ else $X(t_3)[\alpha\leftarrow\pi_2(X(t_1))]$

$Y_\beta{:}OE(\alpha,t_1,t_2,t_3) \Rightarrow$ if $\pi_1(X(t_1))$ then $\lambda x.(Y_\beta(t_1)(Y_\alpha(t_2)(x))$ else $\lambda x.(Y_\beta(t_1)(Y_\alpha(t_3)(x))$
(if β appears free in t_1; $\beta\neq\alpha$)

$Y_\beta{:}OE(\alpha,t_1,t_2,t_3) \Rightarrow Y_\beta(t_2)$ (if β appears free in t_2; $\beta\neq\alpha$)

$Y_\beta{:}OE(\alpha,t_1,t_2,t_3) \Rightarrow Y_\beta(t_3)$ (if β appears free in t_3; $\beta\neq\alpha$)

(if β appears free in more than one of t_1, t_2, t_3, then any of the applicable clauses for Y_β may be used)

(4) $X{:}\lambda\alpha.t \Rightarrow \langle\lambda\alpha.X(t),\lambda\alpha\ x.Y_\alpha(\pi_2(x))\rangle$

$Y_\beta{:}\lambda\alpha.t \Rightarrow \lambda x.(Y_\beta(t)(\pi_2(x)))$ (where $\beta\neq\alpha$)

(5) $X{:}EI(t_1,t_2) \Rightarrow \langle t_1,X(t_2)\rangle$

$Y_\alpha{:}EI(t_1,t_2) \Rightarrow Y_\alpha(t_2)$

(β) Now, to show that pruning is not applicable to the terms extracted by the Dialectica interpretations, we only need to verify that the p-terms t and t′ of the last section yield the same term when given to γ_3. This is straight-forward, since, as we have seen, the Dialectica interpretation and the modified realizability interpretation behave in the same way so long as implication ("$\supset$") is not involved. In particular, we have,

$X(t) = X(t') =$
if $\pi_1(X(t_1))$ then
$\langle 0$, if $\pi_1(X(t_1))$ then $X(t_3)[\beta\leftarrow\pi_2(X(t_1))]$ else $X(t_4)[\beta\leftarrow\pi_2(X(t_1))]\rangle$
else $\langle 1,X(t_5)[\alpha\leftarrow\pi_2(X(t_1))]\rangle$

Appendix B

Content and Form in Proof Manipulation - An Example

There is a sharp contrast between the uses which we have made of proof manipulation methods and the aims for which those methods were originally devised. Namely, normalization, and its predecessor, cut elimination, were developed as tools for use in the mathematical analysis of proofs and provability, whereas we have used them here for the execution and transformation of algorithms. With this shift in aims comes a change in the features of proof systems which are significant. The purpose of this appendix is to illustrate this change by means of an example. In particular, it will be shown in section B.1 that the complexity of the theorems which can be expressed and proved in a formal system - if you like, the "power" or "inferential content" of the system - is not correlated with the complexity of the computations which its proofs can describe. Thus a central feature of proof systems from the point of view of most of proof theory is demonstrably unrelated to the central feature of proof systems for the purposes of computation. In section B.2, the analysis given in section B.1 is extended to normalization with pruning. We begin with a brief discussion of the aims for which proof transformations were developed.

Most work in proof theory has addressed itself to questions which are formulated in terms of *provability* and which do not make direct reference to proofs themselves or to their properties. Questions and results of this kind have a certain generality in that they are independent of the details of how proofs are represented; the differences between the familiar proof systems (such as natural deduction, the calculus of sequents, "Hilbert-style" systems, and so forth) are immaterial from the standpoint of provability - anything that can be proved from given axioms in one system can also be proved in the others. Examples of central notions in proof theory which refer only to provability are (1) the consistency of a theory, (2) the relative "power" of logical principles, and (3) the "proof theoretic strength" of a theory as measured by its ordinal. Of course, the study of questions to do with provability often requires investigation of the details of particular proof systems. Cut-elimination, the ancestor of normalization, was developed as part of just such an investigation; namely the investigation which led Gentzen to his consistency proof for arithmetic from the principle of (quantifier free) induction on the ordinal ϵ_0.

However, formal proofs can also be studied as mathematical objects whose properties are of interest in their own right. For example, the strong normalization theorem for natural deduction [Prawitz 1969] (see chapter 3), and the theorems of [Mints 1977] about the relationships between the "programs" extracted by the various realizability interpretations (see appendix A) are of interest primarily for the theory of proofs (as objects of mathematical study), and not so much for the theory of provability. These results have the common effect of showing that the notion of *the computation described by a proof* is relatively stable under changes of technical formulation. This should be compared to the stability under change of formulation of the notion of a computable function; a stability which constitutes the evidence for Church's thesis.

Kreisel[1969], and Statman[1974] have emphasized that a shift in attention from the theory of provability to the theory proofs leads to a change in the selection of notions and distinctions which are important. As we have said, this appendix is intended to make a similar point in regard to another view of proofs - namely the view of proofs as computational descriptions. The example to be given shortly illustrates the differences between the aims of computation, and the aims of the *theory of provability*. An example of the conflict between the aim of constructing a smooth *theory of proofs*, and the aim of making effective computational use of proofs, has already been seen. Namely, it is essential for the purposes of the stability results mentioned in the last paragraph that attention be restricted to transformations which preserve the extensional meaning of proofs. On the other hand, if one wishes to maximize the computational efficiency of proofs by means of mechanical transformations, then one must use transformations - such as pruning - which change extensional behavior; this was shown by the examples given in chapter 4. Thus the selection of transformations which make for a smooth theory is different from the selection which is best for practical applications. This kind of conflict between the aims of theory and practice is of course common. In the one case general results are what is wanted, and in the other useful techniques - techniques which may or may not have interesting general properties, but which can be profitably applied by the use of human judgement.

B.1 Normalization in successor arithmetic

We proceed now to the example. Let T_S be the the theory which results from restricting the formulation of arithmetic given in section 3.6 to the language which has symbols for successor and predecessor as its only function symbols. Thus the formulas which appear in proofs of T_S will contain (a) zero, (b) "predecessor" and "successor", and (c) "equals" as their only constant, function, and relation symbols, repectively. The lemmas which may appear in proofs of T_S are those of the scheme INDφ of induction, where φ is a formula of the restricted language. From the point of view of the set of provable theorems, T_S is equivalent - modulo a simple translation - to the usual formulation of successor arithmetic. (Predecessor is included as a primitive function because it simplifies the recursive proofs of the induction lemmas). Thus what we have called the "inferential content" of T_S

is exceedingly small. Indeed, from the point of view of the theory of provability, T_S is wholly trivial; it has an elementary decision method by quantifier elimination and a finitist consistency proof. Nonetheless, the computational content of T_S, in the sense of the set of functions which are computed by proofs of $\forall\exists$ formulas, is just the same as that of full arithmetic. This is shown by the following theorem.

Theorem: Let f be a function on the natural numbers which is definable in Gödel's system T of primitive recursive functionals of finite type [Gödel 1958]. Then there is a proof Π_f in T_S of $\forall x\exists y(x=y)$ such that normalization of Π_f computes the function f.

Proof: We will show how to reverse Kreisel's modified realizability interpretation; a map Γ from terms of the typed λ-calculus to proofs of T_S will be described which has the property that the modified realizability interpertation extracts t from $\Gamma(t)$. The map makes use of the correspondence between proofs and λ-terms which was explained at the begining of chapter 3. The end-formula of the proof gotten from a functional f of type "$0\rightarrow0$" will be $\exists x(x=x)\supset\exists x(x=x)$; this proof can be easily transformed into a proof of $\forall x\exists y(x=y)$ which, in the natural sense, computes the same function. The theorem follows since normalization and modified realizability yield the same computations. (See appendix A.)

First of all, we define a map δ from types (of functionals) to formulas by induction on the structure of types. If τ is a type, then $\delta(\tau)$ will be the end-formula of proofs representing functionals of type τ.

(1) Base case: δ: $0 \Rightarrow \exists x(x=x)$

(2) δ: $\tau\rightarrow\rho \Rightarrow \delta(\tau)\supset\delta(\rho)$.

The map Γ is defined by induction on the structure of terms of the typed λ-calculus. For the base case we need to define Γ's behavior on variables and the constant zero. Let $\{x_0,x_1,x_2,x_3 \ldots\}$ be an enumeration of the variables of type τ. Then Γ assigns the proof:

$$\wedge E \frac{[\delta(\tau)\wedge(i=i)]}{\delta(\tau)}$$

to the variable x_i. The second conjunct "$i=i$" (where i is a numeral) serves to label the assumption "$\delta(\tau)\wedge(i=i)$" among all of the other assumptions "$\delta(\tau)\wedge(j=j)$" representing variables x_j of type τ. Next, Γ assigns the proof

$$\exists I\,\dfrac{0=0}{\exists x(x=x)}$$

to the constant zero. The remaining clauses of the inductive defintion are as follows.

(1) succ(t) $\Rightarrow$

$$\exists E\,\dfrac{\begin{array}{c}\Gamma(t)\\ \exists x(x=x)\end{array}\qquad \exists I\,\dfrac{\mathrm{succ}(x)=\mathrm{succ}(x)}{\exists x(x=x)}}{\exists x(x=x)}$$

(2) $\lambda x_i.t$ $\Rightarrow$

$$\supset I\,\dfrac{\begin{array}{c}\Gamma(t)\\ \delta(\tau)\end{array}}{(\delta(\tau)\wedge(i=i))\supset\delta(\tau)}$$

where τ is the type of the term t

(3) $t_1(t_2)$ $\Rightarrow$

$$\supset E\,\dfrac{\Gamma(t_1)\qquad\Gamma(t_2)}{\delta(\rho)}$$

where $\tau\rightarrow\rho$ is the type of t_1, and τ is the type of t_2

(4) $R(t_1,t_2)$ - The types of t_1,t_2 will have the forms τ and $0\rightarrow(\tau\rightarrow\tau)$, respectively. Let F be the formula $\delta(\tau)$. Then the end-formula of $\Gamma(t_1)$ is F, and the end-formula of $\Gamma(t_2)$ is $\exists x(x=x)\supset(F\supset F)$. The proof $\Gamma(R(t_1,t_2))$ uses the induction principle applied to the formula $\varphi(x)\equiv$ "$(x=x)\wedge F$". It will be convenient to use the simpler of the two formulations of induction for φ given in section 3.4, namely, the recursive proof:

$P_\varphi \equiv$

$$
\forall\text{E}\dfrac{
  \forall xy(x=y \vee x\neq y)
}{
  x=0 \vee x\neq 0
}
\quad
\text{SB}\dfrac{
  [x=0] \quad \overset{\Pi_1}{\varphi(0)}
}{
  \varphi(x)
}
\quad
\supset\text{E}\dfrac{
  \wedge\text{I}\dfrac{
    [x\neq 0] \quad \forall\text{E}\dfrac{P_\varphi : \forall x(\ \varphi(x))}{\varphi(\text{pred } x)}
  }{
    x\neq 0 \wedge \varphi(\text{pred } x)
  }
  \quad
  \forall\text{E}\dfrac{
    \overset{\Pi_2}{\forall x(x\neq 0 \wedge \varphi(\text{pred } x) \supset \varphi(x))}
  }{
    x\neq 0 \wedge \varphi(\text{pred } x) \supset \varphi(x)
  }
}{
  \varphi(x)
}
$$

$$
\vee\text{E}\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad}{\forall\text{I}\dfrac{\varphi(x)}{\forall x\varphi(x)}}
$$

We take Π_1 in the above to be:

$$
\wedge\text{I}\dfrac{0=0 \quad \overset{\Gamma(t_1)}{F}}{(0=0)\wedge F}
$$

and Π_2 is,

$$
\forall\text{I}\dfrac{
  \supset\text{I}\dfrac{
    \wedge\text{I}\dfrac{
      x=x
      \quad
      \supset\text{E}\dfrac{
        \wedge\text{E}\dfrac{
          \wedge\text{E}\dfrac{[x\neq 0 \wedge ((\text{pred}(x)=\text{pred}(x)) \wedge F)]}{(\text{pred}(x)=\text{pred}(x)) \wedge F}
        }{F}
        \quad
        \supset\text{E}\dfrac{
          \exists\text{I}\dfrac{\text{pred}(x)=\text{pred}(x)}{\exists x(x=x)}
          \quad
          \overset{\Gamma(t_2)}{\exists x(x=x) \supset (F \supset F)}
        }{F \supset F}
      }{F}
    }{(x=x) \wedge F}
  }{x\neq 0 \wedge (\text{pred}(x)=\text{pred}(x) \wedge F) \supset ((x=x) \wedge F)}
}{\forall x.(x\neq 0 \wedge (\text{pred}(x)=\text{pred}(x) \wedge F) \supset ((x=x) \wedge F))}
$$

This completes the inductive definition of Γ. It is a routine matter to check that

$$\gamma(\Gamma(t)) \sim t$$

where γ is the modified realizability interpretation as described in appendix A, and where "$\sim$" represents interconvertability in the λ-calculus. Thus for each function f of type $0 \rightarrow 0$ which is definable in Gödel's system T, there is a proof Π of $\exists x(x=x) \supset \exists x(x=x)$, such that, for all numerals n, the result of normalizing

$$\supset E\;\dfrac{\exists I\;\dfrac{n=n}{\exists x(x=x)} \qquad \begin{array}{c}\Pi\\ \exists x(x=x)\supset\exists x(x=x)\end{array}}{\exists x(x=x)}$$

has the form

$$\begin{array}{c}\Pi_0\\ \exists I\;\dfrac{m=m}{\exists x(x=x)}\end{array}$$

where m is the numeral for f(n). In order to get a proof Π' of a formula of the form $\forall x\exists y(y=y)$ which computes f, simply take

Π' =

$$\forall I\;\dfrac{\supset E\;\dfrac{\exists I\;\dfrac{x=x}{\exists y(y=y)} \qquad \begin{array}{c}\Pi\\ \exists y(y=y)\supset\exists y(y=y)\end{array}}{\exists y(y=y)}}{\forall x\exists y(y=y)}$$

This completes the proof of the theorem.

The theorem shows that the proofs of successor arithmetic, despite their limited inferential content, are just as computationally expressive as those of full arithmetic. The general reason for this is evident - namely, the behavior of normalization depends chiefly on the structure of the applications of induction principles in a proof, and is insensitive to the mathematical content of the formulas to which induction is applied; this is a sense in which normalization depends on the *form* rather than the *content* of proofs.

As an alternative way of expressing the significance of the theorem, one might say that it demonstrates that normalization is a very bad method for treatment of successor arithmetic proofs. There are after all computation procedures for proofs in this theory which are more efficient in the general case than normalization. For example, since all predicates definable in successor arithmetic are decidable, one can take a proof of $\forall x \exists y \varphi(x,y)$ and an input n and produce an output m with $\varphi(n,m)$ by simple linear search: $\varphi(n,0)$, $\varphi(n,1)$, $\varphi(n,2)$ . . . are tested in turn until a number with the desired property is found. In this case, the proof serves only as a guarantee that the search process will terminate. Thus it may happen that the best computational results in proof manipulation are gotten by making use not just of the form of proofs in the way that normalization does, but also of the mathematical content of the formulas which appear in proofs. (We have already seen an example of this; in chapter 4, the mathematical content of the bin-packing proofs was used in the simplex transformations.) Successor arithmetic is an extreme case, since one does quite well by ignoring the proof altogether except in its role as evidence for the truth its end-formula.

B.2 Pruning in successor arithmetic

In the last section we were concerned with normalization without pruning. The question which we address in this section is: how does the addition of pruning to the set of reduction rules used in normalization affect its behavior in the context of successor arithmetic? Certainly, pruning can make a large difference in the computational efficiency of *some* proofs. In particular, each application of induction in proofs produced by the map Γ of the last section constitutes a redundancy of the kind that pruning removes; in order to verify this, the reader need only note that pruning is directly applicable to the normal form of P_φ for any φ. As a consequence, pruning in this case reduces the complexity of the functions computed by proofs in a drastic way; the functions computed by pruned proofs are describable by use of conditional expressions and the successor function alone.

However, it is possible to modify the proofs produced by Γ in such a way that pruning is no longer of any use. It follows that pruning does not reduce the computational complexity of successor arithmetic proofs in the general case. To start with, consider the clause

(1) succ(t) $\Rightarrow$

$$\exists E\,\dfrac{\begin{array}{c}\Gamma(t)\\ \exists x.(x=x)\end{array}\qquad \exists I\,\dfrac{succ(x)=succ(x)}{\exists x.(x=x)}}{\exists x.(x=x)}$$

in the inductive definition of Γ. Now, since the assumption "x = x" is not used in the second premise of the above proof, the pruning rule for ∃-elimination given in section 2.7 is applicable. However, we may take Γ(succ(t)) to be

$$\exists E\,\dfrac{\begin{array}{c}\Gamma(t)\\ \exists x.(x=x)\end{array}\qquad \exists I\,\dfrac{SB\,\dfrac{[x=x\ ]\qquad succ(x)=succ(x)}{succ(x)=succ(x)}}{\exists x.(x=x)}}{\exists x.(x=x)}$$

instead, and in this case pruning is not applicable. By the same kind of trickery, it is possible to modify Pφ in such a way that pruning is no longer of any use. We will show how this is done in a moment, but first we wish to draw some general conclusions about pruning.

The use which is made of the assumption "x = x" in the proof above is inessential. Further, the fact that it is inessential would be immediately recognized by any person who inspected the proof. (For that matter, any person would recognize with equal ease that "∃x.(x = x) ⊃ ∃x.(x = x)" is a true formula, and consequently perceive the uselessness of the elaborate proofs generated by Γ.) It follows that an analysis of dependencies which is routine for a person may of may not lie within the powers of the formal pruning operations with which we have been concerned. The pruning operations are very sensitive to the formal details of the proofs to which they are applied; two proofs which appear to be essentially identical to a person may nonetheless behave very differently under pruning. Nor is pruning in any sense universal among formal operations for the removal of redundant parts of proofs. One can invent a variety of mechanical transformations on proofs which remove redundancies of one kind or another, but which are useful in circumstances where pruning fails. To take just one example, consider the following operation on proofs of arithmetic:

$$\forall I\,\dfrac{\begin{array}{c}\Pi\\ \exists y\varphi\end{array}}{\forall x\exists y\varphi} \quad\Rightarrow\quad \forall I\,\dfrac{\begin{array}{c}\Pi[x\leftarrow 0]\\ \exists y\varphi\end{array}}{\forall x\exists y\varphi}$$

where the condition for the operation is that x not appear free in φ. This operation, which in a certain weak sense is sensitive to the content of proofs, is effective in reducing the computational complexity of the proofs produced by the new version of the map Γ which we are currently constructing - a map which produces proofs to which pruning is not applicable.

Now, in order to complete the definition of the new version of Γ, we need to modify the proof Π_2 which appears as part of the proof $P\varphi$ given in clause (4) of the definition of Γ. The proof Π_2 appears as part of the proof of the third premise of an $\vee$-elimination inference whose first premise reads "$x=0 \vee x\neq 0$". However, in the normal form of $P\varphi$, no use is made of the assumption "$x\neq 0$" in the proof of the third premise. The reason for this is that no use is made of "$x\neq 0$" in establishing the formula "$(x=x)\wedge F$" in Π_2. However, in the following slightly modified version of Π_2, "$x\neq 0$" is used, and consequently pruning is no longer applicable to$P\varphi$.

$$\forall I\,\dfrac{\supset I\,\dfrac{\wedge I\,\dfrac{\begin{array}{c}\Pi_3\\ x=x\end{array} \qquad \supset E\,\dfrac{\wedge E\,\dfrac{\wedge E\,\dfrac{[x\neq 0 \wedge ((\mathrm{pred}(x)=\mathrm{pred}(x)) \wedge F)]}{(\mathrm{pred}(x)=\mathrm{pred}(x)) \wedge F}}{F} \qquad \supset E\,\dfrac{\exists I\,\dfrac{\mathrm{pred}(x)=\mathrm{pred}(x)}{\exists x(x=x)} \qquad \begin{array}{c}\Gamma(t_2)\\ \exists x(x=x) \supset (F \supset F)\end{array}}{F \supset F}}{F}}{(x=x) \wedge F}}{x\neq 0 \wedge (\mathrm{pred}(x)=\mathrm{pred}(x) \wedge F) \supset ((x=x) \wedge F)}}{\forall x.(x\neq 0 \wedge (\mathrm{pred}(x)=\mathrm{pred}(x) \wedge F) \supset ((x=x) \wedge F))}$$

where Π_3 is

$$
\supset E\,\dfrac{\wedge I\,\dfrac{\begin{matrix}\Pi_4\\ x\neq 0\end{matrix}\quad \wedge I\,\dfrac{\begin{matrix}\Pi_4\\ x\neq 0\end{matrix}\quad \wedge E\,\dfrac{\wedge E\,\dfrac{[x\neq 0\wedge((\mathrm{pred}(x)=\mathrm{pred}(x))\wedge F)]}{(\mathrm{pred}(x)=\mathrm{pred}(x))\wedge F}}{\mathrm{pred}(x)=\mathrm{pred}(x)}}{x\neq 0\wedge(\mathrm{pred}(x)=\mathrm{pred}(x))}}{x\neq 0\wedge x\neq 0\wedge(\mathrm{pred}(x)=\mathrm{pred}(x))}\qquad \forall E\,\dfrac{\forall x\ \ y(x\neq 0\wedge y\neq 0\wedge \mathrm{pred}(x)=\mathrm{pred}(y)\supset x=y}{x\neq 0\wedge x\neq 0\wedge(\mathrm{pred}(x)=\mathrm{pred}(x))\supset x=x}}{x=x}
$$

and finally, where Π_4 is

$$
\wedge E\,\dfrac{[x\neq 0\wedge((\mathrm{pred}(x)=\mathrm{pred}(x))\wedge F)]}{x=0}
$$

Bibliography

Aho, A.V. and Ullman, J.D.[1973], The theory of parsing, translation, and compiling, volume 2, Prentice-Hall, London, pp. 924-925

Barendregt, H.P. [1972] *Pairing without conventional restraints*, (preprint)

Bates, J.L. [1979] *A logic for correct program development*, Ph.D. Thesis, Dept. of Computer Science, Cornell University, August 1979

Beckeman,L., Haraldsson, A., Oskarsson,O., and Sandewall, E.[1976], *A partial evaluator and its use as a programming tool*, Artificial Intelligence Journal 7 pp. 319-357

Bishop, E.[1970], *Mathematics as a numerical language*, Intuitionism and proof theory, Proceedings of the summer conference at Buffalo N.Y., 1968, A. Kino, J. Myhill, R.E. Vesley eds., North Holland, Amsterdam pp 53-71

Borning, A.[1979], *ThingLab - a constraint oriented simulation laboratory*, Ph.D. Thesis, Stanford University Computer Science Department, Technical Report Stan-CS-79-746

de Brujin, N. G.[1970], *The mathematical language AUTOMATH, its usage, and some of its extensions*, Lecture Notes in Mathematics, vol. 125, Springer Verlag, pp. 29-61

Church, A. [1941], *The calculi of lambda-conversion*, Ann. of Math. Studies 6, Princeton, N.J.

Clark, K., and Sickel, S.[1977], *Predicate logic: a calculus for deriving programs*, Proc. of Fifth International Joint Conference on Artificial Intelleigence pp. 419-420

Constable, R.L.[1971], *Constructive mathematics and automatic program writers*, IFIP Congress 1971

van Dalen, D.[1973], *Lectures on intuitionism*, Lecture Notes in Mathematics, vol. 337, Springer Verlag, pp. 1-94

Diller, J.[1979] *Functional interpretations of Heyting's arithmetic in all finite types*, Nieuw Arch. Wiskunde, III. Ser. 27, pp. 70-97

Doyle, J.[1978], *Truth maintenance systems for problem solving*, M.I.T. AI lab technical report AI-TR-419, January 1978

Ershov A.P.[1977], *On the essense of compilation*, IFIP Working Conference on Formal Description of Programming Concepts, Saint Andrews, New Brunswick, vol I., pp. 1.1-1.28

Garey, M.R. and Johnson, D.S.[1979], Computers and intractability, a guide to the theory of NP-completeness, W.H. Freeman, San Francisco, pp. 124-127

Gentzen, G.[1969], *The collected papers of Gerhard Gentzen*, (M.E. Szabo ed.), North-Holland, Amsterdam

Godel, K.[1958], *Ueber eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes*, Dialectica 12, pp. 280-287

Goto S., [1979], *Program Synthesis from Natural Deduction Proofs*, International Joint Conference on Artificial Intelligence, Tokyo

Green, C.C,[1969], *Application of theorem proving to problem solving*, Proceedings of the International Joint Conference on Artificial Intelligence, Washington DC, pp 219-239

Harrop, R. [1960], *Concerning formulas of the type $A \rightarrow A \vee B$, $A \rightarrow \exists x B(x)$ in intuitionistic formal systems*, Journal of Symbolic Logic, vol. 25, pp. 27-32

Hewitt, C.[1971], *Procedural embedding of knowledge in planner*, Proceedings of the International Joint Conference on Artificial Intelligence, London

Howard, W.A.[1980], *The formulae-as-types notion of construction*, in *Festschrift on the occasion of H.B. Curry's 80th birthday*, to appear

Katz, S.[1978], *Program optimization using invariants*, IEEE Trans. Software Engineering, Vol. 4, No. 5, Sept. 1978, pp. 378-389

Kleene S.C.[1945], *On the interpretation of intuitionistic number theory*, Journal of Symbolic Logic 10, pp. 109-124

Kowalski, R. [1974], *Predicate logic as a programming language*, Proc. of the IFIP Congress 1974, pp 569-574

Kreisel, G.[1958], *The nonderivability of $\neg(x)A(x) \rightarrow \exists x B(x)$, A primitive recursive, in intuitionistic formal systems* (abstract), Journal of Symbolic Logic, vol. 23, pp. 456-457

Kreisel, G.[1959], *Interpretation of analysis by means of constructive functionals of finite type, in Constructivity in Mathematics*, North-Holland, Amsterdam, pp. 101-128

Kreisel, G.[1969], *A survey of proof theory II.*, in: Proc. Second Scandinavian Logic Symposium, North-Holland, Amsterdam, pp. 109-170

London, P.E.[1978], *Dependency networks as a representation for modeling in general problem solvers*. Ph.D. thesis, U Maryland Department of Computer Science Technical Report 698

Manna,Z. and Waldinger, R.[1979], *A deductive approach to program synthesis*, Fourth Workshop on Automatic Deduction, Austin Texas, pp 129-139

Martin-Löf, P. [1979], *Constructive mathematics and computer programming*, presented at the Sixth International Congress for Logic, Methodology and Philosophy of Science, Hannover, August 1979

McCarthy, J., et al[1962], *Lisp 1.5 programmer's manual*, M.I.T. Press, 1962

McCarthy, J. and Hayes, P.[1969], *Some philosophical problems from the standpoint of artificial intelligence*, Machine Intelligence 4, American Elsevier, N.Y.

Miglioli, P., and Ornaghi, M.[1980], *A logically justified computing model*, Fundamenta Informaticae, to appear

Mints, G.E.[1977], *E theorems*, J. Soviet Mathematics 8, pp. 323-329

Oppen, D.C.[1979], *Pretty Printing*, Stanford Computer Science Department Report Stan-CS-79-770

Prawitz, D.[1965], *Natural deduction*, Almquist and Wksell, Stockholm

Prawitz, D.[1969], *Ideas and results in proof theory*, in: Proc. Second Scandinavian Logic Symposium, North-Holland, Amsterdam, pp. 235-307

Scott, D.[1970], *Constructive Validity*, Lecture Notes in Mathematics, vol. 125, Springer Verlag, pp. 237-275

Shortliffe, E.H.[1974], *MYCIN, a rule-based computer program for advising physicians regarding antimicrobial therapy selection*, Ph.D. Thesis, Computer Science Dept., Stanford University, in Computer-based medical consultations: MYCIN, New York: American Elsevier, 1976

Shrobe, H.E.,[1979], *Dependency directed reasoning for complex program understanding*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Report TR-503

Stallman, R. and Sussman, G. J. [1977], *Forward reasoning and dependency directed bactracking in a system for computer-aided circuit analysis*, Artificial Intelligence Journal, Oct. 1977

Statman, R.[1974], *Structural complexity of proofs*, Ph.D. Thesis, Department of Philosophy, Stanford University

Statman, R.[1977], *The typed λ-calculus is not elementary recursive*, 18th Annual Symposium on Foundations of Computer Science pp. 90-94

Sussman G.J., and Steele, G.L., [1975], *SCHEME, an interpreter for extended lambda calculus*, MIT AI Memo 349

Takasu S.[1978], *Proofs and programs*, Proceedings of the Third IBM Symposium on the Mathematical Foundations of Computer Science - Mathematical Logic and Computer Science, Kansai

Troelstra A.S.[1973], *Intuitionistic formal systems, in Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, A.S. Troelstra, ed., Lecture Notes in Mathematics vol. 344, Springer Verlag, pp. 1-96

Troelstra A.S.[1973], *Normalization theorems for systems of natural deduction, in: Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, A.S. Troelstra, ed., Lecture Notes in Mathematics vol. 344, Springer Verlag, pp. 1-96

Weyhrauch, R.W. and Thomas, A.J.[1974], *FOL: a proof checker for first-order logic*, Stanford Artificial Intelligence Laboratory Memo AIM-235