THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

MECHANIZING STRUCTURAL INDUCTION

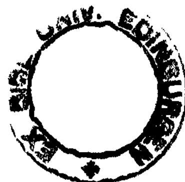
by

Raymond Aubin

Ph D

University of Edinburgh

1976



CONTENTS

ACKNOWLEDGMENTS

SUMMARY

INTRODUCTION

Chapter

1. FORMAL SYSTEM

    1. Syntax

        1. Syntactic Constructs (2)

        2. Introduction of Syntactic Constructs (4)

        3. Inference Rules (8)

        4. Deductions and Proofs (11)

    2. Semantics

        1. Induction (12)

        2. K-recursive Functions (15)

        3. Interpretation (17)

    3. Syntax-Semantics Relation

        1. Soundness (22)

        2. Weak Completeness (24)

    4. Discussion (25)

2. PROOF GENERATION

    1. Search Spaces and Search Strategies (2)

    2. Consistency of Strategies (7)

3. Completeness of Strategies (10)

4. Present Tactics (15)

5. Non-provability Checking (26)

3. INDUCTION VARIABLES AND GENERALIZATION

1. Preliminaries (1)

2. Heuristic Considerations (4)

3. Generalization (10)

4. Selection of Induction Variables (16)

5. Generalization and Strenthening (22)

4. INDIRECT GENERALIZATION

1. Preliminaries (1)

2. Indirect Generalization (7)

1. Search for Mismatches (8)

2. Specialization and Replacement (16)

3. Discussion (22)

5. INDUCTION SUBGOALS

1. Induction Subgoals

1. Extension to Formal System (1)

2. General Method (6)

3. Generation of Induction Conclusions (11)

4. Generation of Induction Hypotheses (16)

2. Use of Induction Hypotheses

1. Replacement (24)

2. Strenthening (29)

6. SIMPLIFICATION AND OTHER STRATEGIES

1. Splitting (1)

2. Contraction (4)

3. Simplification

1. Equivalence and Complexity (7)

2. Selection (14)

CONCLUSION

Appendix

1. TYPE AND FUNCTION DEFINITIONS

2. THEOREMS PROVED

3. SAMPLE PROOFS

1. Distributive Law for Addition and Multiplication with Single Variable (1)

2. Associativity of Multiplication with Accumulator (5)

3. Correctness of Compiling Algorithm for Expressions (11)

4. NOTE ON IMPLEMENTATION

BIBLIOGRAPHIE

A man may be attracted to science for all sorts of reasons. Among them are the desire to be useful, the excitement of exploring new territory, the hope of finding order, and the drive to test established knowledge. These motives and others besides also help to determine the particular problems that will later engage him. Furthermore, though the result is occasional frustation, there is good reason why motives like these should first attract him and then lead him on. The scientific enterprise as a whole does from time to time prove useful, open up new territory, display order, and test long-accepted belief. Nevertheless, _the individual_ engaged on a normal research problem _is almost never doing any one of these things_. Once engaged, his motivation is of a rather different sort. What then challenges him is the conviction that, if only he is skilful enough, he will succeed in solving a puzzle that no one before has solved or solved so well.

Thomas S. Kuhn, _The structure of scientific revolutions_, 2nd ed. enlarged (Chicago: University Press, 1970), pp. 37-38.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to the following people and organizations:

To Robin Milner and Rod Burstall who accepted to become my supervisors half way through my period of study, and whose encouragement and guidance have been invaluable;

To Robert Boyer and J Moore who first aroused my interest in proving program properties, and advised me during the first stages of this research;

To Bernard Meltzer who welcomed me in his research group at the start of my studies;

To Frank Brown, John Darlington, Gordon Plotkin, and Rodney Topor for useful discussions on the topics of theorem-proving, and program proving and manipulation;

To my past and present colleagues of the School of Computer Science and Artificial Intelligence, especially Claude Lamontagne and Pavel Brazdil, for creating such an intellectually stimulating atmosphere;

To Michelle Aubin for her constant personal support and for proofreading this thesis;

To the Commonwealth Scholarship Commission and the Conseil national de recherches du Canada for financial support, and the Science Research Council for computing facilities.

SUMMARY

This thesis proposes improved methods for the automatic generation of proofs by structural induction in a formal system. The main application considered is proving properties of programs. The theorem-proving problem divides into two parts: (1) a formal system, and (2) proof generating methods.

A formal system is presented which allows for a typed language; thus, abstract data types can be naturally defined in it. Its main feature is a general structural induction rule using a lexicographic ordering based on the substructure ordering induced by type definitions.

The proof generating system is carefully introduced in order to convince of its consistency. It is meant to bring solutions to three problems. Firstly, it offers a method for generalizing only certain occurrences of a term in a theorem; this is achieved by associating generalization with the selection of induction variables. Secondly, it treats another generalization problem: that of terms occurring in the positions of arguments which vary within function definitions, besides recursion controlling arguments. The method is called indirect generalization, since it uses specialization as a means of attaining generalization. Thirdly, it presents a sound strategy for using the general

induction rule which takes into account all induction subgoals, and for each of them, all induction hypotheses. Only then are the hypotheses retained and instantiated, or rejected altogether, according to their potential usefulness. The system also includes a search mechanism for counter-examples to conjectures, and a fast simplification algorithm.

INTRODUCTION

The computer provides us with a fast and reliable, let alone flexible, tool for manipulating data. The type of data I am concerned with are sentences in a formal language, and the sort of manipulations I have in mind are drawing inferences, generating proofs, etc. More precisely, this thesis is a contribution to the mechanization of proofs by (structural) induction.

The mainstream of research in mechanical theorem-proving presents only a few projects in which induction has a reasonably important part to play. Darlington (1968) made use of mathematical induction in a second-order resolution theory. Bledsoe's prover (1971) also incorporated an induction rule in a context more oriented toward natural deduction. To my knowledge, he was the first one to have pointed out the generalization problems associated with the mechanical generation of proofs by induction. Brotz (1974) presented us with a theorem-prover for recursive number theory in which induction had the central role to play; the system included also a generalization strategy. Brotz aimed at proving as many theorems as possible only from the axioms, automatically generating lemmas when necessary (and possible). I will come back to Brotz's work on many occasions throughout this dissertation.

However, I am not directly interested in theorem-proving for mathematics: I have a particular application in mind, namely, proving properties of programs. As it turns out, proving facts about recursive programs calls for arguments by induction of one sort or other. Over the last fifteen years, several inductive methods have been proposed for this purpose. Some are more suitable for dealing with real-life programming language features, i.e. loops, assignements, etc.; others cope better with purely functional languages. To some extent, the choice between various inductive methods is a pragmatic question. In this thesis, I concentrate on a functional language without loops or assignments. Two main inductive methods are then available: computational and structural induction. One can hardly avoid mentionning at this point the didactic paper by Manna, Ness, and Vuillemin (1971) on the question.

The ordering used in doing computational induction is a "less defined than" ordering. We consider only monotonic and continuous functions with respect to this ordering; let f be the least fixed point of the equation (i.e. program) $f'=F[f']$. Then to prove the property P(f), it is sufficient to show (1) P("undefined"), i.e. P is true of the totally undefined function, and (2) $P(f')$ implies $P(F[f'])$, for all f'. A Logic for Computable Functions embodying such a rule was mechanized by Milner (1972). At least parts of several involved proofs were carried out mechanically in it: a compiler (Milner and Weyrauch 1972), theorems in various theories (Newey 1973), proofs of LISP programs, interpreter, and compiler (Newey 1975). Von Henke (1975) proposed some heuristics for automating the inductive proofs of at least simple theorems. Finally, a new version of LCF has been designed (Milner, Morris, and Newey 1975; Gordon,

Milner, and Wadsworth 1976) whose main feature is a sophisticated meta-language for writing proof generating strategies.

Structural induction, on the other hand, is of interest in mathematics as well as computer science. This is the method on which the present theorem-proving system is based. Suppose that set S is ordered and every non-empty subset of S has a minimal element. Then to prove the property P(c), for all c in S, it suffices to show P(b), for all b less than a in S, implies P(a), for all a in S. Structural induction is often understood in practice as using a substructure ordering (Burstall 1969, Hoare 1973). Mathematical induction is induction on the structure of natural numbers. Boyer and Moore (1975) wrote an automatic theorem-prover for doing induction on the structure of lists (their domain of lists is defined as the least sets such that nil is a list, and if k and l are lists, cons(k,l) is also a list); it included a generalization tactic similar to Brotz's. A full account of this system is given in Moore's thesis (1973); it has been further developed in Moore (1974). Boyer and Moore claimed that their proof generating methods were relevant to proving properties of LISP programs. The present work is very much in debt to their system, and I will say much more about it in the course of this thesis.

As already hinted at, the general objective of the present research is the design of better methods for automating the generation of proofs by structural induction in a formal system. More precisely, the object of study breaks into two parts: (1) a formal system, and (2) proof generating methods. The distinction between these two aspects should always be made quite clear.

The present formal system allows for a typed language; thus, users can define natural abstract data types and avoid the problem of representing them, say, in terms of lists. The lack of separation between abstract types and their representations was considered to be a weakness of the Boyer-Moore system. As a second objective, I want to make quite precise the ordering which is to be used: I choose a lexicographic ordering based on the substructure ordering induced by type definitions. What ordering the Boyer-Moore induction strategy *precisely* used was not perfectly clear. Types and lexicographic ordering lead to a quite general rule for doing structural induction which also allows non-induction variables to be instantiated in the induction hypotheses.

I also have some goals in mind regarding the proof generating system. The first general objective concerns consistency. I want to be reasonably convincing about the fact that the objects generated by the program are indeed proofs. In my view, this preoccupation does not take a big enough place in many theorem-provers.

As for completeness and efficiency, I set out to investigate three main problems. Firstly, how is it possible to generalize only certain occurrences of a term in a goal? For example, can we get

app(k,app(1,1)) = app(app(k,1),1)

from

app(1,app(1,1)) = app(app(1,1),1)?

Answering this question requires a deeper understanding of generalization than what can be found in Boyer and Moore, or Brotz: they generalize by replacing *all* occurrences of a term by a new variable. My solution derives from thinking of generalization and selection of induction variables as two facets of the same problem.

That is, I generalize only those term occurrences which would be suitable to do induction upon had they been variables. Grossly speaking, a variable is a suitable candidate for induction if it is the first variable which a call-by-need interpreter tries to evaluate (unsuccessfully) in the induction goal. With this method, I can earmark the first and fourth occurrences of l in the above problem, and then, generalize only these to a new variable k.

A further question to be investigated is also pertaining to generalization. One can define functions with arguments which vary within the definitions, besides the recursion controlling arguments. In a theorem to be proved by induction, a variable which occurs in such an argument position can be suitably instantiated in the induction hypothesis in order to match its modified counterpart likely to occur in the induction conclusion. But clearly, this cannot be done when the term in question is not a variable: how is it then possible to generalize such terms? For instance, can we get

```
rev2a(l,k) = app(rev(l),k)
```

from

```
rev2a(l,nil) = rev(l)?
```

Moore (1974) answers this question by introducing in the theorem new function applications whose rators are defined by instantiating a pregiven scheme. I propose a solution which introduces new variables and which I call indirect generalization. In effect, it involves a specialization of the theorem, as a first step; next, the specialized term is used for substituting one subterm of the theorem by another. The new conjecture is still equivalent to the original one, but is in a form which lends itself to an easy generalization. In the above problem, this means first rewriting rev(l) to

app(rev(l),nil), and then generalizing both occurrences of nil to the new variable k.

Thirdly, I consider the following matter: what is a sound way of using the general induction rule? I answer this question by means of a strategy which takes into account all induction subgoals, and for each of them, all induction hypotheses. Only then are the hypotheses retained and instantiated, or rejected altogether, according to their potential usefulness. The choice of keeping or discarding an induction hypothesis depends on the definitions of the functions appearing in the induction goal. The method used by Boyer and Moore to generate induction subgoals left much to be desired as regards consistency.

Other aspects of the work include a technique for finding and using counter-examples. In effect, some strategies may not necessarily yield provable conjectures; generalization is a notable instance. The results of such strategies are accepted only it they cannot be falsified by the method. Finally, another interesting part of the system is a fast simplification algorithm.

Each method proposed in this work is automatic, as is the global system. My goal is actually to minimize the interaction between the prover and the user, and let the system discover for itself the necessary lemmas whenever possible. However, this is not for me a matter of principle: on the contrary, I am not convinced (and was not from the start) of the practicality of purely automatic theorem-proving. The automatic discovery of useful lemmas which are (even mildly) syntactically distant from the goal is very difficult indeed. Nonetheless, it is also clear that we wish to remove as much of the burden as possible from the user's shoulders. One of my

objectives is to see how much further one can go in the direction set out by Brotz, and Boyer and Moore. Furthermore, I think that automatic provers are the best test benches for automatic methods; if these prove to be of any utility, they can be made to profit in a more practical man-machine system.

Cartwright (1976), a contemporary, has also been working on the footsteps of Boyer and Moore. He also felt the need for a richer type structure and a more powerful rule of induction. His rule appears to incorporate some form of generalization in it. Moreover, his verifier is based on a sophisticated simplifier, coupled with interaction with the user.

Chapter 1 of this thesis presents the formal system in which the prover evolves and is self-contained. Chapter 3, 4, and 5 form the core of the proof generating system and bring answers to the three problems stated above. Their reading should required little knowledge of preceding material apart from section 5 of the second chapter. Chapter 2 spells out in general terms the conceptual framework in which the strategies of the prover are set; it also describes a method for finding counter-examples. Finally, chapter 6 deals with other strategies, and in particular, simplification. Appendices contain function definitions and sample proofs; in particular, sections 1, 2, and 3 of appendix 3 exemplify the three problems stated above together with their solutions.

CHAPTER 1

FORMAL SYSTEM

Before going into the question of how proofs can be rigorously generated by machine, we must be quite clear about what a proof exactly is. We must decide about correct sentences, correct inferences, etc. in a language. In a word, we need a formal syntax. At the same time, if we want to do more than just playing with meaningless symbols, we must be precise about what the sentences of our language are intended to signify. In a word, we need a formal semantics. This chapter presents a formal system particularly suitable for talking about structures and for proving theorems by induction on these structures. In the following sections, I will give the syntax and the semantics of this formal system, I will relate these in terms of soundness and completeness, and finally, I will discuss motivational and practical aspects of the system. The general presentation is inspired from Robbin (1969) and Milner, Morris, and Newey (1975).

1.1 SYNTAX

Logicians are very concrete about the formal syntax of a language. They give a number of primitive symbols and then define the set of admissible strings of concrete symbols. I will in general

be more abstract without confusion. On the other hand, I will make use of abbreviations whereby metalinguistic names are used in place of syntactic constructs.

1.1.1 Syntactic Constructs

Type constants

We have an infinite list of primitive constructs iota[1], iota[2], ... called *type constants*.

Type constants will be abbreviated by names. We use the metavariables rho, sigma, tau, maybe with indices, to vary over type constants. An expression like rho* stands for the list rho[1], ..., rho[n] (0=<n) of type constants. The length of the list is denoted by length(rho*), but can usually be inferred from the context.

Variables

We have an infinite list of primitive constructs v[1], v[2], ... called *variable tokens*. A *variable of type tau* is defined thus: if v[i] is a variable token and tau a type constant, then v[i]:tau is a variable of type tau.

Variables will normally be abbreviated by names. We use x, y, z to vary over variables and x*, y*, z* will denote lists of distinct variables.

Constructor constants

We have an infinite list of primitive constructs c[1], c[2], ... called *constructor tokens*. If c[i] is a constructor token and sigma*, tau are type constants, then c[i]:sigma*->tau is a

constructor constant of type sigma*->tau.

The arity of a constructor constant c[i]:sigma*->tau is equal to length(sigma*). We use names to abbreviate constructor constants and we use the metavariable c to range over them.

Defined function constants

This section is analogous to the previous one. We have an infinite list of primitive constructs f[1], f[2], ... called _defined function tokens_; if f[i] is a defined function token and sigma*, tau are type constants, then f[i]:sigma*->tau is a _defined function constant of type sigma*->tau_.

Constructor and defined function constants form the class of _function constants_. The metavariables f, g, h are used to vary over defined function constants and over function constants; the context will make clear which is meant.

Terms

The class of _terms of type tau_ is inductively defined thus:

1. If x is a variable of type tau, then x is a term of type tau
2. If c is a constructor constant of type sigma*->tau and t* are terms of types sigma* respectively, then c(t*) is a term of type tau
3. If f is a defined function constant of type sigma*->tau and t* are terms of type sigma* respectively, then f(t*) is a term of type tau
4. A construct is a term only as required by 1, 2, and 3.

A term which is not a variable will be called a <u>function application</u> (or sometimes more simply, an application). The first and second parts of a function application are given the names of <u>rator</u> and <u>rand</u> respectively; the elements of a rand are the <u>arguments</u> of the application. In the following text, I will freely infix rators whenever possible; the computer program makes use of a different concrete syntax. As already seen above, the metavariables s, t, u, w are used to vary over terms. An expression like f(t*) denotes the function application f(t[1], ..., t[n]) (n=arity(f)); an expression like f*(t*) denotes the list of applications f[1](t[1,1],...,t[1,n]), ..., f[m](t[m,1],...,t[m,n]). An expression like t[s/x] denotes the term resulting from replacing all occurrences of x by s in t; t[s*/x*] denotes t[s[1]/x[1]] ... [s[n]/x[n]]. The fact that s[1], ..., s[n] possibly occur in t may be emphasized by writing t[s*].

1.1.2 Introduction Of Syntactic Constructs

Potentially, we have a countable number of type constants, variables, constructor constants, defined function constants. However, each of these constructs used in the system has to be previously <u>introduced</u> (or defined, or distinguished, or declared, these are all synonymous as far as I am concerned). This introduction is done in a hierarchical manner and serves also the purpose of giving names to the constructs in question. From now on, I will not distinguish between a syntactic construct and its name. This section is really part of the metalanguage.

Type and constructor constants are introduced together in type definitions. A type definition is a pair whose second component is the type constant being defined and whose first component is a list of pairs. Each of these pairs have a constructor constant and a list of type constants. A type definition is admissible only if all syntactic constructs used in the definition (type and constructor constants) have been previously introduced to the exception of the type constant being defined. In the text, I will use a concrete representation e.g.

```
[true: | false: ] -> bool
[zero: | succ:nat] -> nat
[nil: | cons:nat,list] -> list
[atom:nat | consx:sexpr,sexpr] -> sexpr
[nulltree: | tip:nat | node:tree,nat,tree] -> tree
```

The computer program uses another concrete representation.

A type constant is said to be reflexive if it is defined in terms of itself; it is non-reflexive otherwise. (This term borrowed from Milner, Morris, and Newey (1975) is preferred to inductive or recursive.) I will come back to why mutually reflexive type constants are not permitted under the present syntax. We consider that type constants are named by the words e.g. bool, nat, etc. as well as by the whole type definitions.

The type of a constructor constant is immediate from the type definition e.g. ->bool for true, nat,list->list for cons, etc. In practice, I will often omit the parentheses in the terms like true() when it can be done without confusion. By analogy with type constants, we say that a constructor constant c[i]:sigma*->tau is reflexive if tau is in sigma*; it is non-reflexive otherwise. An argument of c[i]:sigma*->tau occurring in the position of tau in sigma* is called reflexion argument; we define non-reflexion

argument by the negative. An immediate predecessor of a list c*(x*) is a list c[1](x[1]*), ..., c[i-1](x[i-1]*), x[i,j], s[i+1], ..., s[n], such that x[i,j] is a reflexion argument of c[i](x[i]*) and s[j] (i+1=<j=<n) is any term; we will see that the choice of the phrase "immediate predecessor" is meaningful.

Variables are not recursive and need not be hierarchically introduced. We say that we declare them: for example, [a | b]:bool, [m | n]:nat, [j | k | l]:list, etc.

Finally, defined function constants are introduced by stages with the help of definitions by cases (Burstall 1969, Hoare 1973). Here are some concrete examples:

```
a=>b : bool <=
cases a [true <= b |
         false <= true]

a & b : bool <=
(a=>(b=>false))=>false

m=n : bool <=
cases m [zero <= cases n [zero <= true |
                          succ(n) <= false] |
         succ(m) <= cases n [zero <= false |
                             succ(n) <= m=n]]
```

They introduce the function constants =>, &, and = for terms of type nat. The computer program makes use of a different concrete syntax.

Abstractly speaking such a definition is a pair. The first component is called the head; it is also a pair having (1) a function application with the function constant being defined as rator and a list of distinct variables as rand, and (2) a type constant. The second component of the definition is either empty (we then say that the function constant is vacuously defined) or it is a case expression. A case expression is either a term or it is a pair formed from a case variable and a list of case clauses. A case

variable is a variable; a case clause has a pattern and a case expression. A pattern is a function application with a constructor constant as rator and a list of distinct variables as rand.

Admissible definitions are as follows: All constructs, apart from the function constant being defined, must have been previously introduced. Case variables must occur in the head of the definitions and cannot appear twice in any chain of nested case expressions; this ensures that definitions are finite. The constructor constants in the rators of the clause patterns of any case expression are precisely the constructor constants of the type of the case expression variable; in a chain of nested case expressions, the clause pattern variables must be all distinct from one another and from the head variables (however, we allow a case variable to recur as reflexion variable in its twin pattern). Arguments in the position of a case variable for the function constant being defined are called <u>recursion arguments</u>; the other arguments are called <u>non-recursion arguments</u>; we also talk of recursion and non-recursion terms. The recursion arguments of a function application are ordered by the order of appearance of case variables from left to right and from top to bottom.

If a case expression is a term, then it is the last of a chain of nested case expressions; the only variables which may occur in this term are the clause pattern variables for the chain and the head variables which are not case variables for the chain. Furthermore, if the function constant being defined appears in the term, then the list of its recursion arguments must be an immediate predecessor of the corresponding list of clause patterns in the chain. Again, I wish to consider that a defined function constant is

named by its whole definition as well as by a word.

Section 3 of appendix 3 shows many type and function definitions as well as variable declarations in a complex situation leading to a compiling algorithm.

1.1.3 Inference Rules

This section expounds the legitimate inferences which can be made in one atomic step. Each one is given as a list of hypotheses separated from a conclusion by a line; hypotheses and conclusions are terms. If we can also infer the conjunction of the hypotheses from the conclusion, we write a double line. We then say that hypotheses and conclusion are interdeducible; alternatively, we say that the rule is reversible. Axioms are inference rules with an empty list of hypotheses.

<u>Truth</u>

```
------
true()
```

<u>Specialization</u>

```
   u
-------
u [t/x]
```

<u>Definition by k-recursion</u>

When a function constant is defined by cases, we can trace in the definition chains of nested case expressions ending with a term. There is an inference rule corresponding to each such chain; together, they form a definition by k-recursion. Consider a single chain. Let f be the function constant being defined; x* be the case

variables in the chain; c*(y*) be the clause patterns in the chain (we have that length(x*)=length(c*)); and t be the term which ends the chain. We apply the substitution [x*/c*(y*)] to the head of the definition by cases and obtain a function application f(s*). We collect all the variables in f(s*) in a list z*; no other variables than z* occur in t. We finally have the inference rule:

```
w [f(s*)[u*/z*] / x]
====================
w [  t  [u*/z*] / x]
```

We get one such rule for each chain of nested case expressions in the definition of f.

In particular, we have for the function constant =>

```
w [true=>s / x]            w [false=>s / x]
===============            ================
w [   s    / x]            w [  true   / x]
```

for the function constant &

```
w [          s & t           / x]
=================================
w [(s=>(t=>false))=>false / x]
```

for a polymorphic equality function constant = of type tau,tau->bool, for every pair of constructor constants c[1], c[2] for type tau,

```
w [c[1](t*)=c[2](s*) / x]
=========================
w [      false       / x]
```

if c[1] is different from c[2], and

```
w [     c[1](t*)=c[2](s*)      / x]
===================================
w [t[1]=s[1] & ... & t[n]=s[n] / x]
```

if c[1] is identical to c[2].

Modus ponens

```
s     s=>t
----------
    t
```

<u>Substitutivity of equality</u>

```
----------------------
u[x/z] & y=x => u[y/z]
```

<u>Induction</u>

```
u[1] ... u[n]
-------------
     u
```

where each u[i] is an implication of the form:

```
u [s[1]* / z*] & ... & u [s[m]* / z*] => u [c*(x*) / z*].
```

There is precisely one u[i] for each list c* of constructor constants for the types of the variables z* respectively. For every u[i], the s[j]*'s are precisely the (uninstantiated) immediate predecessors of c*(x*), and the variables of u different than z* are replaced by distinct metavariables over terms in each antecedent member (although this is not expressed in the above scheme in order to keep notation down). All these metavariables can in fact be instantiated in more than one way. The variables z* are called <u>induction variables</u>.

For example, double induction on the variables m and n of type nat is:

```
u [zero/m] [zero/n]

u [zero/m] [n/n] => u [zero/m] [succ(n)/n]

u [m/m] [s[1]/n] => u [succ(m)/m] [zero/n]

u [m/m] [s[2]/n] & u [succ(m)/m] [n/n]
    => u [succ(m)/m] [succ(n)/n]
-----------------------------------------
                  u
```

where s[1] and s[2] can be any terms.

1.1.4 Deductions And Proofs

A *deduction of a term t from a finite set of terms S* is an acyclic directed graph, with a set of terms T, including t and the elements of S, as set of vertices, with a set of arcs A, and such that:

1. If the terms u[1], ..., u[n], all in T are the initial vertices of n arcs in A directed toward a term u in T-S, then u is an immediate consequence of u[1], ..., u[n] by virtue of an inference rule
2. Term t is the only vertex with no arcs directed away from it.

The term t is called the *conclusion* of the deduction; the terms in S, the *hypotheses*. The *degenerate deduction of t* is the deduction of t from the singleton of t. A deduction which a subgraph of another deduction D is called a *subdeduction of D*.

A *proof of a term t* is a deduction of t from the empty set of hypotheses; the conclusion of a proof is called a *theorem*. In effect, the conclusion of a deduction need not be a theorem nor need the hypotheses. For example,

```
                               ----
        false                  true
      ---------              ---------
      false=true             true=true
  -----------------     -----------------
  (true=>false)=true    (false=>false)=true
  ---------------------------------------
              (a=>false)=true
```

is a deduction of (a=>false)=true from the singleton of false.

1.2 SEMANTICS

Our formal syntactic constructs are intended to denote some objects. I will first study the domain of interpretation of the constructs, and then describe this interpretation precisely. Much of the material has been adapted from Preparata and Yeh (1973) and Cohn (1965).

1.2.1 Induction

We actually want our domain of interpretation to have more structure than being just a collection of sets and we impose an algebraic structure on it. More specifically, our domain is a *heterogeneous* (or many-sorted) *word algebra generated from the empty set*. Such an algebra is a system H=[(S);(c)] which consists of a family of k (1=<k) *carriers* (S) such that each member of (S) is a set, and a family of l (0=<l) *constructors* (c) such that each member of (c) is an n-ary (0=<n) function from n sets in (S) to a set in (S). A constructor which maps into a carrier S is called a *constructor of S*.

The elements of H are precisely those obtained by applying the constructors (c); they are given the name of *structures*. Clearly, in an algebra generated from the empty set, a carrier S is empty only if it has no constructor c from S* to S such that each of S* and S are in (S) and S is not a member of S*. Such a constructor is called a *constant constructor with respect to S*.

Finally, such algebras have the interesting property of being *totally free* from any special identity relation, that is, no non-trivial relation of the form s[1]=s[2] holds in it, where s[1] and s[2] are distinct elements of the algebra. This property is called *unique factorization*. It says in our case that two structures of H are identical if and only if they have been constructed by the same constructor from identical structures.

We define a *substructure relation =<* on each carrier S of H thus: for every s and t in S, s=<t if and only if s=t or s is used in the construction of t. As usual, I will write s<t (read: s is a proper substructure of t) in place of s=<t and s/=t.

Fact

The relation =< is an ordering.

Proof

It is clearly transitive, reflexive, and antisymmetric. ||

We can now talk of the ordered set [S;=<]. An element s of [S;=<] is *minimal* in [S;=<] if no other element of [S;=<] is a substructure of s. The minimal elements of [S;=<] are precisely the structures constructed by applying the constant constructors with respect with S.

Fact

Every non-empty subset of [S;=<] has a minimal element (the *minimum condition*).

Proof

This holds since any element s of [S;=<] has a finite number of substructures by construction. ||

Note however that not all carriers [S;=<] are partly well-ordered since we have e.g. for tree structures infinitely many minimal elements: nulltree(), tip(zero()), tip(succ(zero())), ...

Now given the collection of ordered carriers [S;=<], we define the *lexicographic relation =<<* on S* where S* is the product of not necessarily distinct carriers of H thus: for all s* and t* in S*, s*=<<t* if and only if s[1]=t[1] and ... and s[i-1]=t[i-1] and s[i]=<t[i], for some i (1=<i=<length(s*)=length(t*)).

Fact

The relation =<< is an ordering.

Proof

This holds since =< is an ordering for each S of S*. ||

We let s*<<t* stands for s*=<<t* and s*/=t*, that is, s*<<t* if and only if s[1]=t[1] and ... and s[i-1]=t[i-1] and s[i]<t[i].

Fact

The ordered set [S*;=<<] satisfies the minimum condition.

Proof

This holds since [S;=<] satisfies the minimum condition for each S of S*. ||

We can now assert that the *principle of structural induction* is valid for the ordered set [S*;=<<], that is

```
if, for all t* in [S*;=<<], P(t*)
whenever P(s*), for all s* in [S*;=<<] such that s*<<t*
then P(t*), for all t* in [S*;=<<].
```

In fact, for the moment, I will make use of a weaker induction principle. In any ordered set, we say that an element r is an *immediate predecessor* of an element s in the set, if r is less than s and there is no other element t in the set such that r is less than t and t is less than s (this gives sense to a corresponding notion introduced in section 1.1.2). To obtain the induction principle which we will use at first, one should read "such that s* is an immediate predecessor of t*", instead of "such that s*<<t*" in the above induction principle.

1.2.2 K-recursive Functions

This section studies a class of functions over the carriers of the algebra H.

A function f from S*xT* to S, such that S*, T*, and S are carriers of H, is said to be *defined by k-recursion* if and only if for all lists of k constructors c* of S* respectively, f(c*(x*),y*) is explicitly defined using only:

1. The variables x[1]*, ..., x[k]*, y*
2. The functions lambda y*.f(z*,y*), where z* is an immediate predecessor of c*(x*) in [S*;=<<]
3. Previously defined functions.

Note that an explicit definition is finite. In a sense, this definition scheme says too much and too little. It says too much because, in so far as Peter's results for number theory (1967) are applicable to this system, definitions by k-recursion are reducible to a normal form which does not look like the above definition scheme. But this reduction is a bit artificial and in this system, I

wish to deal with definitions by k-recursion as people would naturally write them. But then the above scheme says too little since it does not cater for course-of-values and mutual recursion. The reasons behind it are essentially pragmatic and I will have the occasion of coming back on them.

We inductively define the class of *k-recursive functions* thus:

1. Constructors are k-recursive functions
2. If f is a function defined by k-recursion from k-recursive functions, then f is a k-recursive function
3. A function is k-recursive only as required by 1 and 2.

Fact

There exists a unique function f from S*xT* to S which satisfies a given definition by k-recursion.

Proof

The proof by induction on the class of k-recursive functions and on [S*;=<<] divides into two parts. We consider all lists c* of constructors of S* respectively.

Existence. For all x* in appropriate carriers, all y* in T*, by induction hypothesis, we have that (1) there exists a z in S such that f(z*,y*)=z for all z* immediately preceding c(x*) in [S*;=<<] and for all y* in T*, and (2) for any previously defined function g, there exists a z in an appropriate carrier such that g(y*)=z for all y* in appropriate carriers. But f(c(x*),y*) is explicitly defined in terms of these functions only. Hence, there exists a z in S such that for all x* in appropriate carriers, for all y* in T*, f(c(x*),y*).

Uniqueness. Suppose some function f′ also satisfies the definition by k-recursion of f for all x* in appropriate carriers and for all y* in T*, then by induction hypothesis, we have that f′(z*,y*)=f(z*,y*) for all z* immediately preceding c*(x*) in [S*;=<<] and for all y* in T*. Hence, f′(c*(x*),y*)=f(c*(x*),y*) for all x* in appropriate carriers and for all y* in T*.

In conclusion, there exists a unique z in S such that f(x*)=z for all x* in S*xT*; so, there exists a unique function which satisfies a given definition by k-recursion. ||

1.2.3 Interpretation

Now that we have a reasonably clear picture of what our domain of interpretation looks like, we can give the intended meaning of our syntactic constructs. Since this section, and the following ones, will mention both syntactic constructs and objects in our domain, the latter will be underlined. Identity between elements of our domain will be written as ==.

An interpretation is a triple <C,M,V> of semantic functions respectively called *classification*, *model*, and *valuation*. These functions map syntactic constructs into semantic objects.

We define the semantic function C (classification) for type constants thus: C assigns a carrier S to each type constant sigma.

In particular, we have that C(bool) is BOOL, the set of truthvalues.

The semantic functions M (model) and V (valuation) for other syntactic constructs in the language are mutually defined:

1. M assigns a constructor c from S* to S to each constructor constant c of type sigma*->tau where C(sigma[i]) is S[i] and C(tau) is S
2. To each function constant f of type sigma*->tau non-vacuously defined by cases, M assigns a function f from S* to S defined by k-recursion, where C(sigma[i]) is S[i] and C(tau) is S; there are precisely as many distinct clauses in the definition of f as there are distinct chains of nested case expressions in the definition of f; if f(x*) is the head of the definition by cases, if z* are the case variables and s* their twinned patterns in a chain, and if term u ends the chain, then the corresponding clause of the definition of f by k-recursion is V(f(x*)[z*/s*])==V(u) where M(f) is f; if f is vacuously defined, M assigns to it a function f from S* to S
3. V assigns an element of S to each variable of type sigma
   that
   such C(sigma) is S
4. V(f(t*))==M(f)(V(t[1]), ..., V(t[n]))
5. M is a model and V, a valuation only as required by 1, 2, 3, and 4.

In particular, we have that M(true) is true and M(false) is false; the meanings of the function constants =>, &, and = are the functions respectively defined by

```
true() => b == b
false() => b == true()

a & b == (a=>(b=>false())=>false()
```

c[1](x*)=c[2](y*) == false(), if c[1] is different from c[2]
c[1](x*)=c[2](y*) == x[1]=y[1] & ... & x[n]=y[n], otherwise.

With the help of the semantic functions C, M, and V, we can obtain a value (i.e. a structure) for any term in our language. For the moment, we will focus our interest on boolean terms. Let B=[(BOOL,S);(true,false,c)] be a heterogeneous word algebra. We say that a term t of type bool is valid in B if and only if V(t)==true() for all values of its vacuously defined function constants and variables.

Before closing this section, there remains a point to be clarified. We have seen, for example, that the meaning of => of type bool,bool->bool is the function => from BOOLxBOOL to BOOL. However, one can legitimately asks whether the function => carries the same information as implication. The question arises because we have a logic of terms only. We first need some definitions. For all functions f from S* to BOOL with S[i] different from BOOL for some S[i] of S*, we define the relation P[f] such that P[f](x*) if and only if f(x*)==true(); such functions f are called predicates. Similarly, for all functions f from BOOL* to BOOL, we define the composition of sentential connectives ("implies", "and", etc.) Q[f] such that Q[f] (P[g[1]](x[1]*), ..., P[g[n]](x[n]*)) if and only if f(g*(x*))==true(); such functions f are called connectives.

Fact

Our interpretation respects truth.

Proof

We have that P[true] is the true relation since true()==true(). ||

Fact

Our interpretation respects falsity.

Proof

We have that P[false] is the false relation since false()=/=true(). ||

Fact

Our interpretation respects implication.

Proof

We want to show that Q[=>] is the sentential connective "implies". This is immediate from the fact that truth and falsity are respected by our interpretation. For example, (true()=>false())==false() if and only if the true relation does not implies the false relation. ||

Fact

Our interpretation respects conjunction.

Proof

This is immediate from the previous results. ||

Fact

Our interpretation respects equality.

Proof

This is the most interesting case. We want to show that (x=y)==true() if and only if x==y for all x and y in a carrier S; in other words, P[=] is the identity relation. The proof is by induction on the family (S) of carriers as hierarchically introduced by type definitions, and on the ordered sets [S;=<]. Suppose c* are precisely the constructors of S; we consider all pairs of constructors c[1] and c[2]. We have that (c[1](x*)=c[2](y*))==true() if and only if (x[1]=y[1] & ... & x[n]=y[n])==true(). But by induction hypothesis, equality is respected for previously introduced carriers and for elements of S preceding c[1](x*) and c[2](y*). So, we have x[1]==y[1] and ... and x[n]==y[n], since conjunction is respected. We finally get c[1](x*)==c[2](y*) because of the unique factorization property of identity. In conclusion, our interpretation respects equality. ||

1.3 SYNTAX-SEMANTICS RELATION

This section studies the relation between the syntax and semantics of our formal system. We are interested in two aspects: soundness (is every theorem valid?) and completeness (is every valid term a theorem?). In fact, we are going to show a much weaker form of completeness: every valid term without vacuously defined function constants and variables is a theorem.

1.3.1 Soundness

The least property which a formal system must have if we want to give any substance to our proving theorems is soundness, that is, we want to make sure that the terms provable in the system are indeed valid.

Fact

If a boolean term t is a theorem, then t is valid.

Proof

The demonstration is by induction on the structure of proofs. We must show that for each rule of inference, if the hypotheses are valid, then the conclusion is also valid.

Truth. We have that V(true())==M(true)()==true().

Specialization. Since u is valid, it is true() for all values assigned to its vacuously defined function constants and variables. In particular, it is true() for V(t) assigned to x for all values of the vacuously defined function constants and variables in t. Hence, u[t/x] is valid.

Definition by k-recursion. By the definition of M and V, the inference rules constituting the definition by k-recursion of a defined function constant f and the clauses of the definition by k-recursion of M(f) correspond precisely. So, for any constituent of the definition of f,

```
w [f(s*) / x]
=============
w [  t   / x]
```

if and only if V(f(s*))==V(t). But, the latter identity holds since we have shown that functions defined by k-recursion are well-defined. Finally, since the identity relation is substitutive, we have that

w [f(s*) / x] is valid if and only if w [t / x] is valid.

Modus ponens. Assume that s and s=>t are valid. Then we have that V(s)==true(), and V(s)==true() implies V(t)==true(), since interpretation respects implication. So, we can deduce that V(t)==true() and hence, that t is valid.

Substitutivity of equality. Since implication, conjunction, and equality are respected by the interpretation, this axiom is valid if and only if V(u[x/z])==true() whenever V(u[y/z])==true() and y==x. But this is precisely an instance of the substitution principle for the identity relation which holds in our domain.

Induction. Two facts should be clear from the start: (1) a list of terms of types tau* is an immediate predecessor of another list of terms (section 1.1.2) if and only if the list of values of the first terms immediately precedes the list of values of the second terms in [C(tau[1]), ..., C(tau[n]); =<<] (section 1.2.1); (2) c[1]*, ..., c[m]* are precisely the lists of constructor constants for types tau* if and only if the values of the terms c[1]*(x[1]*), ..., c[m]*(x[m]*), for all values of the variables, are precisely the elements of [C(tau[1]), ..., C(tau[n]); =<<].

The induction rule is valid if and only if

u[1] and ... and u[n] implies V(u)==true(),

where each u[i] has the form

V(u[s[1]*/z*])==true() and ... and V(u[s[m]*/z*])==true()
implies V(u[c*(x*)/z*])==true(),

under the provisos on c* and s[j]* given in section 1.1.3. But because of the two facts above, the conjunction of the u[i]'s is then equivalent to

V(u)==true(), for all z*,
whenever V(u[y*/z*])==true(), for all y* immediately preceding z*
in [C(type(z[1])), ..., C(type(z[m])); =<<].

Hence, the induction axiom is valid if and only if an instance of the principle of structural induction holds.

This complete the proof. ||

1.3.2 Weak Completeness

The incompleteness theorem of number theory extends to this formal system despite its limited form of quantification (i.e. an implicit outermost universal quantifier for all variables). It is however weakly complete in the sense that every valid term without vacuously defined function constants and variables is a theorem.

Fact

If terms t and s do not contain any vacuously defined function constants and variables, then s=t whenever V(s)==V(t).

Proof

The proof is by induction on the class of terms. If t or s are variables, then the theorem holds vacuously. Let t=f[1](t*) and s=f[2](s*); by induction hypothesis, we have that t[i]=s[i] whenever V(t[i])==V(s[i]). If both of f[1] and f[2] are constructor constants, then by the unique factorization property of equality, we can deduce that f[1](t*)=f[2](s*) whenever V(f[1](t*))==V(f[2](s*)). If at least one of f[1] or f[2] is a (non-vacuously) defined function constant, then by the uniqueness of functions defined by k-recursion, we also have that f[1](t*)=f[2](s*) whenever V(f[1](t*))==V(f[2](s*)). This

completes the proof. ||

As a matter of fact, the converse also holds. This justifies what will be called evaluation, that is, the repeated application of the k-recursive definition rule to a term t without vacuously defined function constants and variables, until it cannot be applied any more. When t contains vacuously defined function constants or variables, we then talk of symbolic evaluation.

The weak completeness theorem is a corollary of the above proposition.

Fact

Every valid term without vacuously defined function constants and variables is a theorem.

Proof

In other words, we want to show that if V(t)==true(), then t is a theorem. Assume V(t)==true(); then by the above fact, t=true(). But this is equivalent to t=>true() and true()=>t; hence, by modus ponens, true() and t are interdeducible. In conclusion, t is a theorem. ||

1.4 DISCUSSION

One objective of this formal system was to start from a small base in order to achieve a great degree of uniformity as regards e.g. induction. We explicitly introduced the set of truthvalues BOOL with the type definition [true: | false:] ->bool. An inference rule was given about the constructor constant true. We also introduced the function constants => and = by k-recursion. Moreover, we gave for

these function constants the rules of modus ponens and substitutivity, which, one must admit, have very much of the same flavour. The function constants true, false, =>, and = constitute a small enough number of basic constructs which should really be considered as logical symbols.

This system looks very much like one of the Hilbert type. There is nothing special about this: it just seems to be a consequence of wanting to start from a small base. However, it is possible to raise the level of this base and derive from it many other rules. Not only is it possible, but it is also desirable, and our mechanical theorem-prover will, in fact, deal with an amplified system. An amplification of the system is always achieved by deriving a rule of inference. If the derived rule is a deduction (or indeed a deduction scheme) but not a proof, then I will specifically talk of derived rule; otherwise, I will say that the system is amplified by a theorem. In the next chapter, I will deduce the derived rules available to the present prover. Modus ponens is not part of them: it is only used as a building block in the derivation of more involved rules. The set of derived rules is fixed and cannot easily be amplified without reprogramming. However, theorems can be added with more flexibility, even if we will also start with a fixed set of them.

To start with, we define the following function constants in addition to those defined previously (we redefine & as it is actually defined in the program):

```
a & b : bool <=
cases a [true <= b |
         false <= false]
```

```
a v b : bool <=
cases a [true <= true |
         false <= b]

not(a) : bool <=
cases a [true <= false |
         false <= true]

cond(a,b,c) : bool <=
cases a [true <= b |
         false <= c]
```

Note that cond is of type bool,bool,bool->bool; however, in the rest of this text, I will use the same function constant for all conditionals of type bool,tau,tau->tau. The function constants =>, &, =, v, not, and cond constitute all the connectives used in practice in the system.

A fixed theorematic amplification has been brought to the basic system; it consists essentially of a set of equalities used to put a term in normal form. In effect, presenting the various strategies of a theorem-prover with terms in normal form is of paramount importance if any reasonable degree of uniformity is to be achieved. These equalities are put into action by means of a rule derived from the substitutivity of equality. The normal form which has been chosen is a conjunction of implications whose antecedents are conjunctions and consequents are disjunctions. More graphically, a term in normal form satisfies the following scheme:

```
(a[1,1] & ... & a[1,k[1]]) => (c[1,1] v ... v c[1,m[1]])
& ... &
(a[n,1] & ... & a[n,k[n]]) => (c[n,1] v ... v c[n,m[n]])
```

(0=<n, 0=<k[i], and 0=<m[i]), where a[i,j] and c[i,j] are terms without connectives and conditionals.

I will come back in chapter 6 on the precise equalities which are used in normalization. Suffice to say for now that they have been directly inspired from Ketonen's dialect (1945) of Gentzen's sequent calculus (1955). This variant is quite convenient since contrary to Gentzen's original version, the rules are all reversible. This is why we can use equalities in the present case. So, in a sense, we can say that at least the propositional part of this amplified calculus behaves like the sequent calculus. To justify this choice, I must anticipate a little bit on chapter 2. The niceness of the sequent calculus resides in the subformula principle: for every proof in normal form, each step is no more complex than the conclusion; and Gentzen proved that every proof can be put in normal form. Even if the subformula principle does not hold locally for non-logical inference rules, it allows nevertheless a very natural backward search strategy which is easier to mechanize. This was first recognized by Wang (1960) and interest in such systems has recently been renewed, e.g. Brown (1976a).

Even if sequent calculus systems are particularly well adapted to automatic theorem-proving, there is an element of unnaturalness about them precisely because forward search is practically prohibited. Hence, in the framework of interactive theorem-proving, it seems to be more advisable to build upon a natural deduction system (Gentzen 1955, Prawitz 1971, and for an application, Milner, Morris and Newey 1975) wherefor forward and backwark search can equally be used; a reasonably directional forward search is difficult to mechanize, but the user is there to help.

In addition to this basic logical amplification, one can add further equalities and equality schemes to the set of theorems, which equalities are used subsequently for simplification purposes.

The next point I would like to discuss concerns types. The decision of using a typed language is of great pragmatic importance. The main advantage of it, which concerns k-recursive definitions as well as inductive proofs, is the separation of abstract structures from their concrete representations. Knowing that any of our structures can be coded as, say, natural numbers may be theoretically appealing, but we are chiefly concerned with practical feasibility, and it suffices for us to know that such a coding can be done. In this respect, we follow the steps of the structured programming school (Dahl, Dijkstra, and Hoare 1972) which advocates the abstract development of programs in which types have a particularly important role to play. In a second stage, one can represent his abstract program in the concrete programming language and with the concrete structures of his choice.

So, one should look at our k-recursive definitions as abstract programs, or alternatively as concrete programs in a programming language where the user could define new types as we can do it in our formal language. I will not be involved with the representation question. Similarly, proofs of programs can sometimes be broken with profit by (1) proving an abstract program, and (2) showing that it simulates a concrete one. Our proofs about k-recursive functions can be said to fulfill part one.

Boyer and Moore (1975) used the other approach of explicitly representing datatypes in terms of their unique type list. This one is defined as [nil: | cons:list,list] ->list. Their representation functions can be defined thus: the booleans true and false are represented by the function R[bool] such that:

```
R[bool](true)=cons(nil,nil)
R[bool](false)=nil;
```

the natural numbers are represented by a function R[nat] such that:

```
R[nat](zero)=nil
R[nat](succ(x))=cons(nil,R[nat](x))
```

In a version not reported in the literature, they also represented symbolic expressions and binary numbers. I think that Boyer and Moore had two difficulties. Firstly, they lacked a general and formal framework to talk about representation. Secondly, they did not have a general induction strategy which would, at least implicitly, have derived induction rules for the various new types as they were represented in the language. This is one reason why their trial with symbolic expressions was only partly successful: properties about them were proved by list induction. I must say however that they had in the end an ad hoc mechanism to make list induction behave like mathematical induction when the induction variable was a number.

In the event of these difficulties being overcome, I would still prefer a typed language. One reason is that types are helpful in preventing and detecting meaningless constructions. For example, I have always felt uneasy about Boyer and Moore writing functions to reverse a number or give its length. But even if one tries to avoid such definitions, mishaps are always possible. Every term used by this prover is type checked. This is quite easy to fulfill: a term

of type tau is well-typed if and only if it is a variable of type tau, or if it is a function application whose rator is a function constant of type sigma*->tau and whose arguments are well-typed terms of types sigma* respectively. Experience shows that typing mistakes are relatively frequent and even such a simple type checker proves quite helpful in detecting them.

Another undesirable feature of a type-free language is the all too frequent necessity of specifying the type of a term by means of an explicit type predicate. For example, in the case of Boyer and Moore, these predicates may look like:

```
isbool(true)=true
isbool(false)=true

isnat(zero)=true
isnat(succ(x))=isnat(x)
```

Adding type predicates to the statement of a theorem can considerably increase its complexity. Furthermore, type predicates cannot be considered on the same level as the other assumptions e.g. by the induction strategy. This is bound to lead to increased difficulty in the design of proof generating computer programs. For example, Boyer and Moore have to write

```
isnat(x) & even(x)
=> double(half(x))=x.
```

Even when type predicates need not be mentioned in the original theorem, they often have to be introduced later after a generalization has taken place. In his thesis (1973), Moore gave a nice procedure for doing this automatically. However, in a later stage, he added some <u>metalinguistic</u> conventions by which the letters i to n would denote natural numbers. The above theorem could then be abbreviated by

```
even(i)
=> double(half(i))=i
```

The automatic generation of type predicates was also dropped.

In conclusion, one can justly say that the result of the Boyer-Moore experiment points toward a typed language.

This decision, however, is not without problems. This leads me to consider the reasons why function definitions are written using case expressions rather than conditional expressions as in Boyer and Moore. McCarthy (1966) advocated the use of conditional expressions in writing recursive function definitions. One can point at three distinct roles that such expressions play: (1) they can replace patched together definitions; (2) they can represent definition schemes (by primitive recursion, k-recursion, etc.); (3) because of their special semantics, they can act as minimization operators in general recursive function definitions. (The phrase "patched together definitions" is taken from Peter (1967) and is preferred to "definitions by cases" or "definitions by composition" used in other quarters.) We can eliminate usage 3 since we do not deal with general recursive functions. The distinction between 1 and 2 is in direct anology with the distinction one can make between type predicates and any predicates. Boyer and Moore used conditionals for 1 and 2; I will use case expressions for 2 and reserve conditionals for 1.

Definitions by cases are much more elegant, much less prone to error, and more in the spirit of disciplined programming. They have been advocated by Burstall (1969) and Hoare (1973) particularly well. I will also make the point in chapter 6 that expressions with conditionals are simplified less efficiently. A more profound reason for using case expressions is that in this typed language, we simply

cannot say everything we want with conditionals. The problem stems from the fact that in general, there is no way of writing discriminators using conditionals only. A discriminator is a predicate which tells whether the last constructor applied in the construction of a given structure is a given constructor or not.

However, it can be done for Boyer-Moore lists. In the following primitive recursion scheme:

```
f(l,k*) : list <=
cond(null(l),
     g(k*),
     h(car(l),cdr(l),k*,f(car(l),k*),f(cdr(l),k*))),
```

the discriminator null is easily defined by

```
null(l) : list <= l=nil.
```

However, we cannot write a similar scheme for trees (which have been defined as [nulltree: | tip:nat | node:tree,nat,tree] ->tree), since we now have countably many tips and countably many nodes. Had we represented trees as lists, a discriminator istip could have been defined by conditionals since tree tips would then have constituted a certain subclass of lists.

In summary, by using a system where objects of various types can be referred to directly without being represented by e.g. lists, we have lost the possibility of defining some functions by conditionals. But this is more than compensated by the use of case expressions.

As a final matter regarding types, the definition of mutually reflexive types has been barred by the metalinguistic rules laid down in the previous sections. There is no special point about this, since mutually defined types have a meaning in the domain of interpretation. The reason for this exclusion is pragmatic: no

strategies have been studied concerning such types.

The desire for working with natural types and leaving the representation question aside is analogically reflected in our scheme of definition by k-recursion. As mentioned earlier, in as much as Peter's results (1967) carry up to this formal system, our scheme says too much: there is a much more compact normal form. But I will not be disturbed by this: as for representation, I wish to ignore the question of reducibility of k-recursive functions. More will be said about this in chapter 5 where a weak form of course-of-values recursion, namely recursion from several bases, will be explicitly introduced. As for mutually defined functions, they have been ignored for pragmatic reasons just like types: relevant strategies have not been explored in any depth for them.

Finally, one must have noticed the absence of quantifiers in this formal language. We can say for a start that quantifier-free languages do exist. Resolution is a notable example in automatic theorem-proving and the elimination of quantifiers cuts down an important source of complexity. Formulas in such languages are in Skolem normal form wherefor, roughly speaking, existentially quantified variables are replaced by function applications whose rators are new function constants called Skolem constants. This skolemization procedure is in fact applicable to a formula in prenex normal form consisting of a list of quantifiers followed by a quantifier-free formula. Looking from the opposite direction, in the theorems to be proved, it is the universally quantified variables which are replaced by new applications. In the present formal language, one can view all variables in a term as being universally quantified by quantifiers whose scopes are the whole term.

Consequently, _no_ variables are replaced by Skolem function applications in a term. On the other side, in a conjecture, _all_ variables should be replaced by Skolem applications so that there is no need to do it explicitly: we just have to remember that these variables cannot be instantiated. And as a matter of fact, there is no inference rule in the system which allows such instantiations. There is an exception, however: in proofs by induction, any variables other than the induction variables can be specialized in the induction hypotheses. This has been built into the induction rule and reflects the implicit outermost presence of existential quantifiers for induction steps. In summary, one can think of the boolean terms of this language as first-order formulas in Skolem normal form with universal quantifiers only, except for the non-induction variables in induction hypotheses.

This is quite sufficient to serve our purposes. Our k-recursive definitions are, in a sense, programs and the theorem-prover proves properties about these programs. With such an application in mind, the usefulness of existential quantifiers appears to be limited. Moreover, the presence of free variables in conjectures leads in general to explosive search strategies in view of the numerous ways in which free variables can be instantiated. Our use of such variables is limited to induction for which meaningful instances of variables are relatively easy to find.

CHAPTER 2

PROOF GENERATION

Once the legal moves of theorem-proving game have been decided upon, one is left with the more challenging problem of finding winning strategies. Three objectives have to be met:

1. To respect the rules of the game - a question of consistency
2. To win as often as possible - a question of completeness
3. To use as little resource as possible in doing so - a question of efficiency.

This chapter starts by expounding a framework for talking about proof generation. The following sections deal with the questions of consistency and completeness for strategies. This leads to introducing in general terms the proof generating methods used by this prover and how they interrelate. Finally, a technique for trying to find counter-examples to conjectures is explained. The first sections borrow some material from Kowalski (1972) and Milner et al.'s work on LCF (Milner, Morris, and Newey 1975; Gordon, Milner, and Wadsworth 1976); the last ones, from Boyer and Moore (1975) and Brotz (1974).

2.1 SEARCH SPACES AND SEARCH STRATEGIES

In the previous chapter, I gave a definition of the notion of proof of a term t. The general problem I am now addressing is that of finding such a proof for a given term called a _goal_ (or conjecture). For any term, this problem may have zero or more solutions; if it has more than one solution, these may be more or less complex, according to some measure of proof complexity. However, the question whether a goal has a proof or not is undecidable in general, e.g. the specialization rule can be used in countably many ways.

The set of all graphs of terms in the language is called the _search space_, and an algorithm which generates a subset of the search space is called a _search strategy_. A search strategy is said to be _consistent_ if it generates deductions only. If a deduction has the initially given goal as conclusion, then it is called a _reduction_; a _solution_ is a proof of the goal. Any reasonable strategy will have embodied in it a termination-with-solution condition, and a search strategy is _complete_ if it finds a proof of the conjecture whenever there exists one.

Since we are, in fact, searching for proofs (only proofs can be solutions), it may seem sufficient, and perhaps more natural, to define the search space as the set of all proofs. But then, it would only be possible to generate proofs from proofs, whereas the present definition allows consistent search strategies to generate proofs from any deductions, and in particular, from reductions. Hopefully, this gain in flexibility will outweigh the fact that there are many more deductions than proofs.

I will now give a non-deterministic consistent search strategy (a sort of generalized British Museum Algorithm) in the context of which the actual strategy used by the present prover will be explained later.

By this strategy,

1. If a solution has been generated, we terminate; otherwise, we do either step 2 or step 3
2. If t is any term, we create the degenerate deduction of t and we go to step 1
3. If D[1], ..., D[n] are generated deductions of t[1], ..., t[n] respectively, and if a hypothesis s of a generated deduction D, distinct from D[1], ..., D[n], immediately follows from t[1], ..., t[n] by means of an inference rule, we generate the deduction obtained from D[1], ..., D[n], and D by adding an arc from each of t[1], ..., t[n] to s; we go to step 1.

This algorithm can be improved by noticing the following: if two deductions share the same conclusion, and if the hypotheses of one are contained in the hypotheses of the other, it is not necessary to generate them both. One may wish to retain, perhaps, only the simplest one, according to some complexity measure.

Other questions arise when we get less abstract and more concrete. When a deduction is generated, is it allowed to share some of its structure with already generated deductions, or is it constructed afresh? Are identical deductions actually identified? How easily can the redundancy mentioned above be removed, concretely speaking? I will leave this matter aside for the moment.

More important is the question of selecting for generation one deduction among all those generable in one step of the strategy. Such immediately generable deductions are called *candidates* and can be represented by a quadruple <<D[1], ..., D[n]>, D, s, I>, such that hypothesis s in deduction D immediately follows by inference rule I from conclusions t[1], ..., t[n] in deductions D[1], ..., D[n], D being a generated deduction distinct from the generated deductions D[1], ..., D[n].

The election of a candidate is usually done in several stages, that is, the number of candidates is gradually reduced by choosing some <D[1], ..., D[n]>, D, s, and I in some order until one generable deduction is left. Actually, it is often the case that several candidates meet given criteria, and then all of them may be generated.

A preliminary selection of candidates which is frequently (and often implicitly) made is that of a direction. A *direction* is a set of candidates such that <D[1], ..., D[n]> or D (inclusively) have certain properties. Two traditional problem-solving directions are the *forward* (or bottom-up, or synthetical) direction wherefor D[1], ..., D[n] are proofs, and the *backward* (or top-down, or analytical) direction wherefor D is a reduction. So, working forward means starting with axioms and making deductions by means of the inference rules until the goal is deduced; working backward means starting with the goal and reducing it to subgoals (hypotheses of the reduction) until none are left. The terms forward and backward can be justified by looking at the history of the solution: subdeductions generated forward have been grown in the direction of the arcs with respect to D[1], ..., D[n], while those generated

backward have been expanded against the direction of the arcs with respect to D.

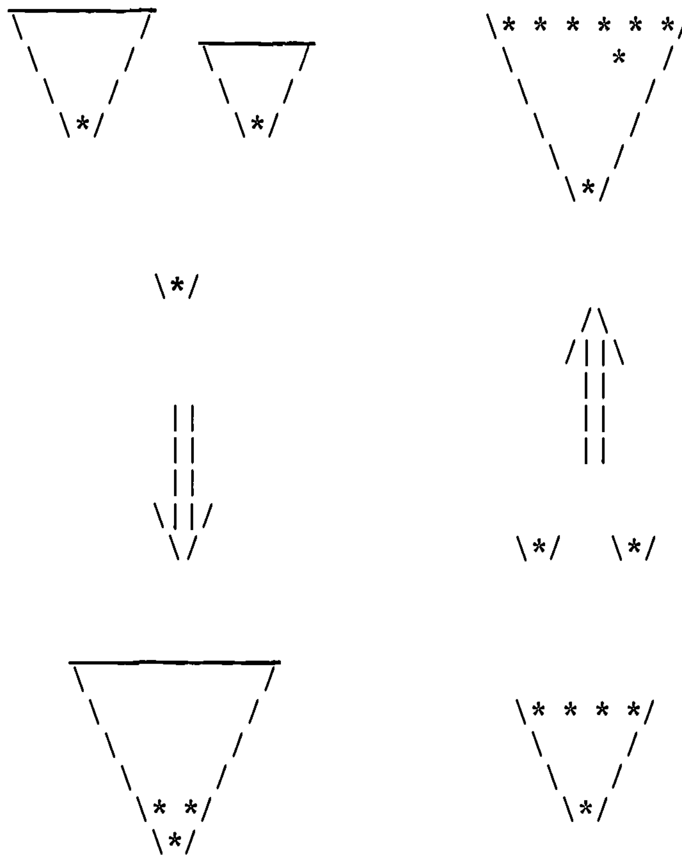


Fig. 2-1. Forward and backward generation.

One direction or another is more appropriate for searching the space of different formal systems. For example, the search space of a resolution-based system is generated forward. The negation of the original term to prove is added to the set of hypotheses, and the goal is the false term. Working backward from falsity would be a rather awkward thing to do. Bi-directional search, however, is quite usual with systems based on a natural deduction calculus. One can make deductions from the assumptions by means of the elimination rules, and reduce the goal by means of the introduction rules; both directions intersect with terms having predicates as rators. Finally, with systems founded on the sequent calculus, assumptions

are explicitly kept in the antecedent of the sequent, and all rules are introduction rules (in the antecedent or consequent). These characteristics make them very suitable for backward search.

One can also have a more open mind about directions. For instance, D[1], ..., D[n] may not necessarily be proofs, and D, not necessarily a reduction, but they may be thought to be subdeductions of a solution. Of course, this condition is hard to detect.

As already mentioned, the formal system was basically amplified to model a sequent calculus more closely than any other system: we prove theorems rather than refuting their negations as in resolution; however, there is no assumption making mechanism as in natural deduction. The rule of modus ponens can be seen as a structural rule. Finally, the similarity shows up quite clearly in the choice of the normal form for terms. Consequently with what was said, our search strategy will be generally backward, although it will occasionally take another direction. Preference for this direction is based on heuristic rather than theoretical grounds. That is, only practice, and informal more than formal considerations of completeness and efficiency have presided over the choice of a direction.

So, for all admissible candidates, D will be a reduction, and since the search is unidirectional, it is clear that D[1], ..., D[n] will be degenerate deductions. The strategy will choose D, then s, then I, and finally D[1], ..., D[n]. In other words, it selects one of the subgoals (s) which the original goal has been reduced to, and replace it by a certain number of further subgoals (the terms of D[1], ..., D[n]).

This is a quite standard search. From this view point, the choice of s given D can be called an and-choice since it is necessary to eventually select all subgoals of D and its descendants, if these are to be subdeductions of a solution. The choice of D, or the choice of I and D[1], ..., D[n], given D and s, can be called an or-choice since it is sufficient to select only one way of solving a goal, if the resulting deduction is a subdeduction of the solution.

I will close this section on two remarks. I talked above of forward/backward search, of forward/backward generation of the search space. This is an inexact, if commonly accepted, description. A consistent search strategy generates deductions from deductions, and in this sense, is unidirectional, the goal being the generation of a solution. Only the solution can be interpreted as having been generated forward, backward, etc. by looking at its history. This leads to another point: the notions of deductions, search space, search strategy, etc. are relative to what may be called inference operators. In this section, our inference operators have been the inference rules of our formal system.

2.2 CONSISTENCY OF STRATEGIES

At first sight, the question of consistency of search strategies seems hardly worth mentioning: it is an obviously necessary property to have if any substance is to be given to our claim of generating proofs. As it turns out, however, there are practical difficulties involved with maintaining the consistency of a certain approach to backward search. These difficulties essentially come from the fact that deductions are generated against the

direction of the inference rules (or against the direction of the arcs in the deductions) with respect to the reduction D.

In broad terms, a tactic is a search strategic procedure which takes a goal and delivers one or more subgoals. A tactic is consistent if given some hypothesis s in a reduction D, it selects a rule I and degenerate deductions D[1], ..., D[n], and creates a new reduction such that s follows from the conclusions of D[1], ..., D[n] by means of rule I. However, in actual practice, the provers of Boyer and Moore (1975) and Brotz (1974), for example, do not explicitly check that the application of their tactics is always consistent. Hence, it is not impossible that from a goal which is not provable, a tactic yields subgoals which are all provable. Of course, in general, the method a tactic uses to reduce goals to subgoals will be the inverse of some inference rule. But one must prove this to be the case for each tactic, and as we will see, this may be quite difficult to achieve at the implementation level.

So, it makes sense to talk of degrees of consistency for search strategies, and this assessment is quite important as regards the confidence one should put into the proofs found by a given prover. If alleged proofs are not to be machine-checked, it is all important for the terms output by the prover to be readable; this is one of the reasons why the normal form used by Boyer and Moore (1975) with conditionals has been abandoned here in favour of a form using implications, conjunctions, and disjunctions.

Generating and checking can be used jointly together (see Milner et al. in LCF). Nevertheless, the present prover uses a less satisfactory approach, retaining the method of Boyer and Moore (1975) and of Brotz (1974). It will be shown for each (abstract) tactic,

that for any input, i.e. a hypothesis s in a reduction D, and for any output, i.e. the new reduction generated from D with some D[1], ..., D[n], s does follow from the conclusions of D[1], ..., D[n] by an inference rule I. The rule I need not necessarily be primitive; it may be a derived rule. This solution is less satisfactory because the demonstration has to be done for each tactic; in the case of Milner et al., any inconsistency in any tactic is automatically trapped at the checking stage.

However, proving the consistency of tactics (and inference rules) at an abstract level is not sufficient at the implementation level; in effect, our confidence in the proofs generated by consistent tactics does not go beyond the confidence we have in the correctness of their implementation. Quite clearly, the proofs are going to be generated by a computer program. So, the consistency question can be raised in a more concrete, but just as important, fashion: are the programs implementing tactics (and inference rules) correct? With Milner's approach, one has to convince oneself only of the correctness of the programs implementing the primitive inference rules in order to be satisfied with the consistency of any search strategy. And these programs are bound to be quite simple. In the present case, as one may guess, it would be necessary to prove the correctness of the implementation of all tactics (written in POP-2 (Burstall, Collins, and Popplestone 1971)) in order to achieve the same degree of confidence. This is an exacting task. However, this may sometimes prove more efficient with small tactics which have been extensively experimented with and have reached a quite definite form. At any rate, one will have to be contented with abstract consistency justifications of the present tactics and to make a bigger act of

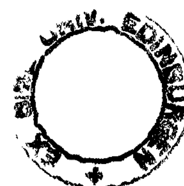
faith in the solutions displayed by the program.

In conclusion, the requirement that a search strategy should follow the rules of the game as stated at the beginning of this chapter has been shown to be far from a trivial question. It is an important matter to consider in order to judge the degree of confidence one should put in the proofs given by such and such theorem-prover, especially when the theorems are not obvious and the proofs, long.

2.3 COMPLETENESS OF STRATEGIES

In the previous section, we discussed the problem of consistency of the search strategy. We wanted to ensure that a tactic would not yield provable subgoals from a goal which is not provable, or if it did, that this non-provable step would be detected sooner or later.

This section discusses a different problem, that of completeness. It has been said that a complete search strategy finds a solution whenever there exists one. However, I will not discuss the existence and nature of a complete uniform strategy for backward search. Each of my tactics has been devised to help find solutions for certain classes of problems and does so with more or less generality and success. So, it seems more appropriate to talk of degrees of completeness, and to consider the matter rather informally. It is not so unreasonable to proceed in this way since a (formally) complete search strategy may be seen as incomplete in practice if it takes an unbearably large amount of resource to obtain the average solution. Some may wish to see the question as one of



efficiency, but this is simply putting the matter off.

One problem associated with backward search regarding completeness can be seen as the inverse of the consistency problem: that is, we want to ensure that from a provable goal, a tactic yields only provable subgoals. Note that there is no such question with forward search since the generated deductions are always proofs. Obviously, this is only a necessary condition: if a tactic does not have this property, the generated deduction cannot be a subdeduction of a solution. On the other hand, this property is clearly not sufficient in order to find a solution if there is one. This necessary condition about completeness is reasonably easy to fulfill. We have said that a tactic can be generally regarded as using an inference rule backward. Three situations can arise:

Firstly, the inference rule is actually reversible, or in other words, the conjunction of the hypotheses, and the conclusion are interdeducible. This is true of our definition rule, for instance.

Secondly, the rule is not generally reversible but it is possible to find some (syntactic) constraints on its application so as to make it reversible for those cases. For example, a weakening rule whereby a hypothesis is added to the antecedent of an implication is reversible in the context of induction. In other words, if a goal is valid and we apply an induction tactic to it, then no matter how many induction hypotheses are discarded in an induction goal, it will still be valid.

Thirdly, the reversibility of a rule is not absolutely certain in a given context, but has some reasonably high probability of being so. Generalization of a term to a variable when the term occurs on both sides of an equality or implication is an example of this. Then, one must use some external help to make sure that each particular instance of the rule is reversible; this is the role played by the search for counter-examples to be studied in section 5 of this chapter.

Hence, for all three cases, we can attain a fairly high degree of confidence in the provability of the subgoals produced by a tactic. Quite clearly, reversible rules should be exploited whenever possible; on the other side, rules with only a small chance of reversibility in some context should not generally be applied in tactics.

An immensely more difficult problem about completeness is that of finding a sufficient set of tactics to get all the proofs we are interested in. This is shared with search strategies using any direction of search. As already mentioned, I will not look at the question from a theoretical view point. If we take, for example, the ability of discarding induction hypotheses, it seems quite clear that, used indiscriminately, this tactic cannot in general give all the proofs we want, even though the subgoals may all be provable. So, the biggest challenge for backward search is the design of relevant tactics. In the next section, I will introduce the tactics used in the present prover.

How good is backward search? For a start, there are more than one method of working from a goal. Polya (1965) suggests three such methods:

1. Trying to solve a more ambitious goal (or to prove a stronger conjecture, or to show a sufficient condition, or to generalize)
2. Trying to solve a logically equivalent goal (but thought to be simpler according to some complexity measure)
3. Trying to solve a less ambitious goal (or to prove a weaker conjecture, or to show a necessary condition, or to specialize).

Methods 1 and 2 correspond to what we have called working backward, and if the more ambitious or equivalent subgoals are fulfilled, then the original goal is solved too. We cannot draw the same conclusion for method 3; the solution of the less ambitious problem does not immediately give the key to the original problem, but can provide some help toward its solution.

We will see that method 3 will be quite helpful in attempting to establish the non-provability of a subgoal. Brotz (1974) makes also use of this third method to discover and prove subsidiary goals. These may be obviously useful to the proof, but on the other hand, extremely difficult to generate by conventional subgoaling tactics. This has long been recognized in mathematics by the use of lemmas (in the original sense).

So, a pure subgoaling method does not appear to be sufficient in practice. The addition of previously proved results has actually often been part of theorem-provers (except that Brotz discovers them). Boyer and Moore (1975) started off with a fixed, but

non-independent set of axioms, so that some provable facts were readily available to their backward strategy. An idea along the same line is that of the textbook method, or the so-called extensible prover (Brown 1976b) whereby an arbitrary number of (non-independent) facts, perhaps constrained to be of a certain form, can be added to the system in some hierarchical fashion. The prover of Moore (1974) also incorporates this facility; it presents the advantage of allowing conditional facts to be added. Cartwright (1976) can also use such facts in his verifier.

The aim of the present prover is chiefly to experiment with generalization tactics combined with induction, and not necessarily to show as many theorems as possible. However, previously proved equalities and equality schemes can be used in amplifying the basic system, and if some fact needed in a proof cannot be discovered automatically by the prover, I have no scruple in adding it to the system. This will be discussed in greater detail in section 3 of chapter 6.

In conclusion, it appears that a practically complete search strategy for a wide class of interesting theorems in a theory is a long way off, for any direction of search. I have not at all discussed the advantages and disadvantages of forward search. As regards backward search, one thing seems certain: the strategy needs to make use of prior experience. This experience may be given to it in a more or less interactive way (Brown 1976b) or it may be discovered for itself (Brotz 1974).

However, should one aim at practical theorem-proving, the strategy of letting the theorem-prover build its own experience is unlikely to pay off in the short or middle term. For efficiency as well as completeness reasons, it is more realistic to allow the addition (in a structured way) of previously proved facts, either, so to speak, off-line, between the proofs, or better still, in a more interactive mode of interaction.

2.4 PRESENT TACTICS

In the previous sections, I have only mentioned the question of concrete representation for the deductions and the search space without going into it. Indeed, a choice of representation is not directly pertinent to the consistency and completeness of the search strategy, but rather to its efficiency.

Deductions, as abstract structures, have been defined as graphs. However, in the present prover, they are represented as trees, this being a first approximation. This entails some obvious redundancies. The vertices of the trees are term occurrences rather than terms, or in other words, different occurrences of a term in a deduction are not actually identified. Consequently, the same subgoal may have to be solved more than once on different branches of the tree. This is a well-known redundancy of tree representation.

There are some advantages to this representation. Identifying two occurrences of the same term is resource consuming, and if it cannot be done efficiently, it may be better to solve the same subgoal more than once, when the situation occurs. Moreover, the representation facilitates sharing. It has been mentioned that

if two deductions share the same conclusion, and if the hypotheses of one are contained in the hypotheses of the other, it is not necessary to generate them both. This means in our particular case of unidirectional backward search, that a reduction can actually be represented as the list of its hypotheses. And this is the representation that has been adopted in this prover. The and-choice of a hypothesis for generation of a new deduction is done on last in-first out basis.

So, the generated search space is a set of term lists. However, when the present strategy makes an or-choice, it never reconsiders it; it never comes back to a choice point after having generated part of the search space, and having been forced to stop for reason of failure or other. In other words, the strategy does not backtrack. (This, I hope, clears up a misunderstanding concerning the Boyer-Moore prover about its doing no search; of course, it does some search (any prover does), but it never backtracks.) A consequence of this technique is that quite a bit of resource must be put into selecting the best possibility at each or-choice point. For example, finding a correct generalization, or chosing the right set of induction variables, or the right instantiations of induction hypotheses, all these are quite resource consuming. There is an obvious trade-off here, because backtracking is also expensive. I dare say that for the problems considered, the present approach has paid off.

Now, since the strategy starts off with the reduction composed of the single initially given goal, and makes a unique or-choice when a new deduction is generated, we have that all generated reductions are subdeductions of the last one to have been

generated; and since or-choice are never reconsidered, it is sufficient to retain only the last generated reduction at any stage. So, the generated search space is represented by a unique reduction, which is itself represented by a stack of hypotheses which is called a <u>goal stack</u>.

What has been said about or-choice must actually be put into perspective, since the formal language contains a defined disjunction connective v which may or may not be split using the tactics corresponding to the derived rule

$$\frac{t}{t \vee s} \qquad \frac{s}{t \vee s}.$$

The present strategy does not split disjunctions thus, and consequently can be said to actually delay its decision concerning some possibilities which composed an or-choice. So, some backtracking is allowed to take place in a hidden and limited form.

To summarize what has been said up till now, the present strategy takes a reduction in the form of a goal stack, it selects the top of the stack, andappliesto it a tactic which produces zero or more subgoals; these are pushed onto the stack. A solution is found when the stack is empty.

It remains to be seen which tactics are used and when. The overall structure of the search strategy is very similar to that of Boyer and Moore and of Brotz. It is actually quite simple and can be described thus:

1. If the goal stack is empty, then we exit: all subgoals have been solved; otherwise, we pop up the top of the stack and assign it to a program variable, say, t

2. We simplify t as much as possible, using both definitions, and previously proved equalities and equality schemes
3. If t is a conjunction, we split it in as many subgoals as there are conjuncts, and we push them down on the stack; we pop up the top of the stack and reassign it to t
4. If t is an implication with some equalities in its antecedent, we try to use these to make substitutions in the other members of the implication; some equalities used in the process, especially induction hypotheses, are dispensed with; we simplify the result and reassign to t
5. If contraction is applicable to t (a sort of inverse of substitutivity), we simplify the result and reassign it to t
6. If t is the term false, then we exit: a subgoal is not solvable
7. If t is the term true, then we go back to 1: a subgoal has been solved
8. We select from t a list of variables to do induction upon and in the process, we do any necessary generalizations
9. We push down onto the stack the subgoals resulting from doing induction on the variables selected in t with their induction hypotheses appropriately instantiated; we go to 1. ||

A thorough investigation of these tactics, especially from the points of view of completeness and efficiency, will be done in subsequent chapters. In the final part of this section, I would rather like to investigate their consistency (from an abstract point of view), that is, what rules of inferences can the tactics be said to be the inverses of.

I will proceed by building derived rules in a hierarchical way such that the proof of a rule may involve rules derived earlier as well as primitive rules. Some of the rules may not actually be used for their own sake but simply as building blocks for more useful and important rules corresponding to tactics in the prover. In the end, I will point out which rules correspond to which tactics.

Conjunction

```
t     s
-------
 t & s
```

Proof

```
              ----------------
              x => (y => y & x)
              ----------------
   s          s => (t => t & s)
   ----------------------------
t        t => t & s
---------------
      t & s
```

This derivation makes use of a theorem which, we assume has been previously proved.

<u>Weakening</u>

```
     t
  ------
  s => t
```

Proof

```
     ----
     true             t
  ----------     ---------
  false => t     true => t
  ------------------------
           x => t
           ------
           s => t
```

Among the numerous variants of this rule which can be derived by using properties of & and =>, the following are mainly used:

```
      t => s                t => s
  ----------            ----------
  u & t => s            t => s v u
```

<u>Equality substitutivity (form 1)</u>

```
  w[s/x]      t=s
  --------------
      w[t/x]
```

Proof

```
                            ----------------------
  w[s/x]    t=s             w[z/x] & y=z => w[y/x]
  ------------              ----------------------
  w[s/x] & t=s              w[s/x] & t=s => w[t/x]
  ------------------------------------------------
                 w[t/x]
```

With this rule, all equalities about implication, conjunction, disjunction, and equality itself, to name a few, can be used in a proof. As an immediate instance of this, a sort of inverse of the present rule can be derived using the symmetry of equality:

```
w[t/x]      t=s
--------------
    w[s/x]
```

<u>Equality substitutivity (form 2)</u>

```
t=s => w[s/x]
-------------
t=s => w[t/x]
```

Proof

```
                                                  ---------------------
                                                  w[z/x]  & y=z => w[y/x]
                         ------                   ---------------------
                         x => x                   w[s/x]  & t=s => w[t/x]
                        --------          -------------------------------
t=s=>w[s/x]    t=s=>t=s                   t=s => (w[s/x]  & t=s => w[t/x])
----------------------                ------------------------------------
t=s=>w[s/x]  & t=s=>t=s               t=s=>w[s/x]  & t=s=>t=s => t=s=>w[t/x]
--------------------------------------------------------------------------
                       t=s => w[t/x]
```

Many variations of this rule can be obtained if some properties are taken into account. In particular, using the symmetry of equality, it is possible to get a rule for which the hypothesis and the conclusion are interdeducible. The same frame of derivation can also be used to yield a substitutivity rule for conditional equalities, i.e.

```
u => w[s/x]      u => t=s
------------------------
       u => w[t/x]
```

<u>Equality substitutivity (form 3)</u>

```
     t=s
-------------
w[t/x]=w[s/x]
```

Proof

```
                      ---
                      x=x
                 ------------
                 w[s/x]=w[s/x]
          -------------------
          s=t => w[s/x]=w[s/x]
          -------------------
s=t       s=t => w[s/x]=w[t/x]
--------------------------
        w[s/x]=w[t/x]
```

Substitutivity of general relations

We can extend the notion of substitutivity with profit to any binary relation. It is then possible to derive rules which are generalizations of those given for equality under the three previous headings.

A relation @ is *left-substitutive for term w in variable z* if and only if w[y/z] & y@x => w[x/z] holds; symmetrically, a relation @ is *right-substitutive for term w in variable z* if and only if w[x/z] & y@x => w[y/z] holds.

Modelling on equality, we can derive for left-substitutive relations, the rules:

```
w[s/z]     s@t              s@t => w[s/z]
--------------              -------------
   w[t/z]                   s@t => w[t/z]
```

and for right-substitutive relations, the rules:

```
w[t/z]     s@t              s@t => w[t/z]
--------------              -------------
   w[s/z]                   s@t => w[s/z]
```

If the relation @ is symmetric, then these rules are reversible.

A substitutivity rule

```
     s@t
 ------------
 w[s/x]@w[t/x]
```

can also be derived for a relation @ if it is reflexive, and if it is left-substitutive for w[s/x]@w[z/x] or right-substitutive for w[z/x]@w[t/x] in z.

These rules should be quite helpful in dealing with relations like "less than or equal to", or "greater than" on natural numbers, but I have not concentrated on this. I have limited myself to implication. Indeed, the relation => is left-substitutive and right-substitutive for u=>z and z=>u respectively in z; this is equivalent to saying that => is transitive. So, the rules which will be used are:

```
(s=>t) => (u=>s)              (s=>t) => (t=>u)
----------------              ----------------
(s=>t) => (u=>t)              (s=>t) => (s=>u)

u=>s      s=>t                t=>u      s=>t
--------------                --------------
     u=>t                          s=>u.
```

The last two are, in fact, cut rules. Since => is reflexive, the third form of substitutivity can be derived for it, but it is not very helpful.

These rules can be relaxed by adding conjuncts to the antecedents of implications and disjuncts to their consequents, by means of the weakening rule; together, they justify the consistency of the tactics given above.

The simplification tactic uses not only the primitive rules of definition by composition and k-recursion, but also the substitutivity rule (form 1).

The conjunction rule justifies the splitting tactic.

The substitutivity rule (form 2) verifies the tactic by which an equality in the antecedent of an implication is used to make substitutions in other members of the implication. Moreover, if the equality is then discarded, the strengthening tactic which does it is based on the weakening rule.

The contraction tactic makes use of the substitutivity rule for equality (form 3).

The rule which shows the consistency of the generalization tactic is a primitive one: specialization. In the more complex cases, however, use may be made of the cut rules as intermediate steps.

Finally, the consistency of the induction tactic is established by the induction rule, and also by the weakening rule, if some induction subgoals are strengthened by removing one or more induction hypotheses.

From this, it is possible to conclude that the strategy is consistent on an abstract level. Needless to say that this consistency is not necessarily preserved at the implementation level, as pointed out earlier.

2.5 NON-PROVABILITY CHECKING

It has been said that if a reduction was to be part of a solution, the tactic used in its generation should deliver provable subgoals from a provable goal. This is ensured if the rule which justifies a tactic is reversible in all or some well-defined contexts. But clearly, this is far from being the case for all

tactics given above. Generalization is a notable example.

A well established method of increasing the credibility of subgoals is trying to find counter-examples to them. If such a counter-example is found, then the subgoal is not provable; otherwise, we cannot be absolutely sure that it is provable, but we can have a great confidence that it is so, if the search for a counter-example has been reasonably extensive.

The addition of such a technique to mechanized theorem-proving goes back to Gelernter (1963) and his geometry-theorem proving machine. It has recurred since in various other theorem-proving systems; Brotz uses it in his induction-based prover. Boyer and Moore (1975), however, did not incorporate it.

With a strategy where no explicit backtracking takes place, like in the present prover, it is of vital importance that tactics do create provable subgoals. Consequently, I judge as quite essential the addition of search for counter-examples in such theorem-provers. We will see that some problems cannot be solved without it.

Most people have a semantic approach to this question. That is, they present the problem as looking for a set of values in the domain of interpretation which can falsify the conjecture. It is a minor point, but I prefer a syntatic point of view. This is made possible in the present context because of the explicit representation of truthvalues in the language, and because of the weak completeness theorem. Whether a boolean term without vacuously defined function constants variables is true or false is a decidable question. Talking syntactically seems more appropriate since the decision is taken by using the inference rules of the system.

The third of Polya's methods of working from a goal was to solve a less ambitious goal; it was mentioned that this is not sufficient to obtain a solution but can bring some help. This is what happens here. The help required is an answer to the question: should I believe that such subgoal is provable? The way it is answered is by replacing all the variables in a conjecture by terms containing no variables (using the specialization rule), and then deciding whether or not this instance is true (using the definition and substitutivity rules).

We have to find which terms to use in specializing. Our first aim should be to find an enumeration of all terms of a given type built up only from constructor constants, that is, terms mapping directly onto values. Actually, giving a complete enumeration is not such a good idea. For two reasons. Firstly, it seems more natural to provide a parameterized enumeration. We call a _structure of type tau_ a term such that all its subterms of type tau have a constructor constant as rator and all its subterms of other types are variables. Take, for example, the following definition of trees: [nulltree: | tip:nat | node:tree,nat,tree] -> tree. A complete list of all trees would have to take into account that trees are defined in terms of natural numbers. So we may get something like:

```
nulltree,
tip(zero),
node(nulltree, zero, nulltree),
...,
node(tip(zero), zero, tip(zero)),
tip(succ(zero)),
...
```

This list does not reflect a natural order on trees, like size or level; which natural number is at each node should be irrelevant. The way in which the carriers of our domain are generated demands to generate parameterized structures by increasing level. The _level of a structure of type tau_ is defined as the longest chain of nested reflexive constructor constants for type tau in the structure. The _reflexivity of a constructor constant of type tau_ is defined thus: the non-reflexive constructor constants have reflexivity 0; the reflexive ones have reflexivity equal to the number of their arguments of type tau. So, for trees, nulltree and tip are of reflexivity 0 and node, of reflexivity 2. The level of a tree is then the longest chain of nested occurrences of the constructor constant node in it. We generate all trees of level 0, then all trees of level 1, etc.:

```
nulltree,
tip(n),
node(nulltree, n, nulltree),
...,
node(tip(n[1]), n, tip(n[2])),
...,
node(nulltree, n, node(nulltree, n[2], nulltree)),
...
```

Of course, when these structures are used for checking the non-provability of a conjecture, the specialized goal may still contain variables and the specialization has to be iterated until the goal is ground, i.e. contains no variables. This process terminates because types are hierarchically defined.

The second reason why an enumeration is not desirable as such is that, in general, we want to generate the (n)th element of the list, but without generating all n-1 previous ones. The approach taken toward a solution to this problem is to find in terms of which

substructures indexed by n[1], ..., n[m] is the structure of index n derived, for n[i]<n (1=<i=<m), and to recursively apply the same procedure to each of n[1], ..., n[m], until all substructures have non-reflexive constructor constants. For instance, the (n)th tree will be node(<the (n[1])th tree>, n[2], <the (n[3])th tree>), for all n[i]<n (i=1,3); the question is to find n[1] and n[3] given n.

The algorithm utilizes two subprocedures which I will describe briefly. One of them generates all developments of length n of a natural number l. A development of a number l is a list of natural numbers less than or equal to l, which includes l. The other procedure gives the number of structures of level l of type tau.

<u>Number of structures of level l of type tau</u>

If l is equal to 0, then the number of structures of level l is equal to the number of constructor constants of reflexivity 0 in the definition of tau. Otherwise, set a variable n to 0. For each reflexive constructor constant c in the definition of tau, and for each development l[1], ..., l[k] of l-1, where k is the reflexivity of c, we compute the product of the number of structures of levels l[1], ..., l[k], and add it to n. The final value of n is the result sought for. ||

For example, let N[l] be the number of trees of level l, we have:

```
N[0] = 2
N[1] = N[0]xN[0]
N[2] = N[1]xN[0] + N[1]xN[1] + N[0]xN[1]
N[3] = N[2]xN[0] + N[2]xN[1] + N[2]xN[2] + N[1]xN[2] + N[0]xN[2]
...
```

These procedures are used over and over again by the main algorithm, so that it may be more efficient to store a result once it has been computed in order to look it up later.

The main algorithm is really used to reduce the problem of finding the (m)th structure of type tau to that of finding the (m)th structure of level l for that type. The method is by successive approximation. For the purpose of this algorithm, we say that the (m)th structure in the enumeration is the structure indexed by m-1; so, the first structure is structure 0.

<u>Structure of index n</u>

If the definition of tau has no reflexive clauses and if m is greater than or equal to the number of non-reflexive clauses, then we terminate with the undefined result.

Otherwise, we find the level of structure m by successive approximation. We set a variable n to the number of structures of level 0 for type tau, and we set a variable l to 0. Then, while n is smaller than or equal to m, we reduce m by n, increment l by 1, and reset n to the number of structures of level l for tau.

When final level l is found, we use a subalgorithm to find structure m of level l.

Subalgorithm

If l is 0, then we return the structure made from the (m+1)th non-reflexive clause of the definition of tau. Type constants in this clause are replaced by variables of the same type.

Otherwise, we have to find the top-level reflexive constructor constant of structure m, and which substructures of which levels to put in its reflexion argument positions. We proceed again

by successive approximation, using developments.

For each reflexive constructor constant c for type tau, we generate all developments of l-1 of size equal to the reflexivity of c. The numbers in these developments represent possible levels of substructures, as we have seen. For each development l[1], ..., l[k], we compute the product p of the number of structures of level l[j], for all j (1=<j=<k), for type tau. If p does not exceed m, then we reduce m by p, and we keep on searching. Otherwise, we have completed the first phase: the structure looked for has c as top-level constructor constant and its reflexion arguments are substructures of levels l[1], ..., l[k].

It remains to find out which substructures exactly these are. For each level l in l[1], ..., l[k], we successively divide m by the number of structures of level l, and get a quotient q and a remainder r. On the one hand, we use this subalgorithm recursively to find substructure r of level l; on the other hand, we reduce m by q and keep on going until all substructures have been found.

This completes the algorithm. We can fill in the reflexion argument positions of constructor constant c with the substructures just found, and enter variables of appropriate types in the other positions. ||

Note the generality of these algorithms. If you read "composition" in place of "development" throughout, you obtain a method of enumerating structures by order of increasing size. The size of a structure of type tau is the total number of reflexive constructor constants for tau in the structure; a composition of a number s is a list of natural numbers, the sum of whose elements is s.

These algorithms give us a flexible means of finding counter-examples to a conjecture, that is, finding structures to specialize the variables of a conjecture with. In his arithmetic prover, Brotz chooses these values to be combinations of 0, 1, and other random numbers. We need a way of selecting structures which will work for all types.

The aim is to reject a goal as quickly as possible if it is not provable. For this purpose, structures with top-level constructor constants of reflexivity 0 are an obvious choice for specialization; very often, these constitute the border-line cases for which the goal is not provable (the opposite can also occur but less often, it seems). These are, in fact, the structures of level 0. We also include the structures of level 1; they are also often border-line cases. For example, succ(zero) is an identity element for the function constant * (times); the number list of length 1 is always ordered, no matter what its element is. From there, the idea of using other random, or arbitrarily big, structures is rejected: it is simply too inefficient. Even factorial of 5 is quite long to compute. So, we limit ourselves to adding to the set of specializations of a type the first structure of level 3.

To summarize, the structures used in specializing each variables of the conjecture are all those of level 0 and 1, and the first structure of level 3.

This does not add up to many natural numbers, but for more complex types, the number of structures grows exponentially with respect to level. For example, to check a conjecture with two variables of type tree, we use seven possible instances for each of the variables, which already makes 49 cases... No good solution has

been found to curb this explosive amount of work, other than increasing the efficiency of the various programs involved in checking.

Firstly, the structures used for checking are generated once for all when each type is defined. When they are needed, they are not even copied, but the variables are simply renamed in them. Secondly, not all variables are specialized at one time. The mechanism for successive specialization is already present, since as we have seen specialization itself may introduce variables because structures are parameterized. The variables which are selected at each stage play a key role in the simplification of any term; they are called primary variables and will be studied in the next chapter. The hope is that simplification will eliminate some variables which then will not have to be specialized, thus reducing the number of cases to examine. Thirdly, every effort was put into the design of a efficient simplification algorithm. This will also be discussed later.

Despite all this, checking for the non-provability of subgoals is a necessary but quite resource-consuming strategy.

CHAPTER 3

INDUCTION VARIABLES AND GENERALIZATION

The induction tactic is the centre around which all other tactics gravitate. It has been divided into two distinct parts: (1) the selection of a list of variables to induce upon and (2) the generation of the induction subgoals, given these induction variables. Chapter 5 will treat the latter and this chapter will concentrate on the first aspect. Actually, I submit that selection of induction variables and generalization are intrinsically linked together; so, both will be studied in the following sections. This new look on generalization opens up the possibility of replacing only certain occurrences of a subterm in a goal by a new variable; in particular, this subterm may be a variable itself.

3.1 PRELIMINARIES

What is the problem to start with? We have a goal which we wish to solve by induction. So, we have to find an instance of the induction rule on the basis of which, directly or indirectly, the induction tactic will generate induction subgoals. The first crude step to take, and in the absence of any other clues, is to select some list of variables occurring in the goal to play the role of induction variables. Of course, choosing any list will be grossly

inadequate, in general. Some lists of variables may not lead to subgoals which are any easier to solve. But what is meant exactly by an induction subgoal being more or less easy to solve? We need some criterion. It will be the following one postulate: the more closely the induction hypotheses match the induction conclusion, the more easily the subgoal will be solved. This principle will be consistently followed throughout the rest of this work, and in particular, will be applied to the selection of induction variables.

However, there is a sense in which doing induction on variables is only partly adequate. Indeed, nothing prevents us from deriving a rule for doing what can be loosely phrased as induction on any _term occurrences_ in the goal; this can be achieved by combining our original induction rule together with the specialization rule. I want to propose that it makes sense and that it is useful to think in terms of doing induction on term occurrences. But why is this derived rule hardly ever made use of practice? I think that the answer is simple. In the normal context of induction on variables, the rule of induction is reversible; consequently, the induction tactic can be safely used without checking. However, if we allow induction on any term occurrences, the new rule loses its reversibility property and the provability of the subgoals is no more guaranteed.

Since we naturally use the induction rule backward, it is more convenient to break the tactic into the two constituents: (1) generalization of term occurrences to variables, using the inverse of the specialization rule, and (2) induction on the resulting variables. The subgoals obtained by generalizing must be checked, while the induction tactic can always be used safely. This is the

approach which will be adopted in this prover. Note, however, the new complexion that generalization now takes: we generalize those term occurrences which we think are suitable for doing induction upon. In conclusion, generalization and selection of induction variables are two sides of the same coin.

Furthermore, in as much as Kreisel's (1965) and Prawitz's (1971) results carry up to our formal system, we can assert that doing generalization is necessary in proofs by induction. In other words, no complete backward search strategy can do without it. Prawitz's illustration of this fact is simple and worth reporting briefly.

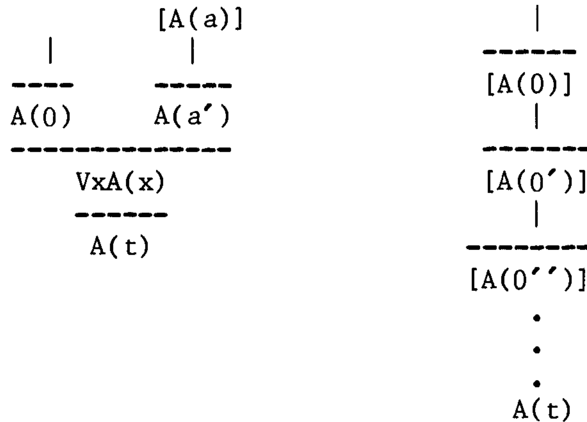
He first gives reductions for derivations in the framework of a natural deduction system. For example, the right derivation below is a reduction of the left:

```
   |                   |
 ------                |
  A(a)                 |
 ------                |
 VxA(x)                |
 ------              ----
  A(t)               A(t)
```

(The V-elimination rule corresponds to our specialization rule.) Such reductions lead to a theorem whereby every derivation reduces to a normal form. When the system is extended to first-order Peano arithmetic, the rule of induction can be formulated as a V-introduction rule (a formula within square brackets is an assumption):

```
         [A(a)]
 A(0)     A(a')
 --------------
     VxA(x)
```

Is there a normal form theorem for this extended system? The answer is no. If t denotes a number (i.e. t contains no variables), then we have the following reduction from the left to the right:



However, when t contains a parameter (i.e. a variable), the reduction is clearly not possible. In other words, if we look at the proof backward, the generalization performed in passing from A(t) to VxA(x) cannot be eliminated. So, generalization is inherent to the backward generation of proofs by induction.

3.2 HEURISTIC CONSIDERATIONS

We now have to tackle the problem from the heuristic point of view. That is, we have to find some relevant principles to preside over the selection of induction variables and generalization.

In the first place, we must take into account the basic objective of optimizing the match between induction hypotheses and conclusions. This is met by using the analogy which exists between recursion and induction: the k-recursive definition rule gives the value of a function in terms of its values for preceding values of its recursion arguments in an ordered set, while the induction rule asserts the validity of a boolean term, assuming its validity for

preceding values of its induction variables in an ordered set. If the induction variables are chosen so that the k-recursive definition rule becomes applicable to an induction subgoal conclusion, when it was not applicable to the induction goal, then we stand some chance of seeing at least part of the induction hypothesis matching at least part of the conclusion. The heuristic importance of this idea was first put in evidence by Boyer and Moore (1975). Reinterpreted in our terms, the theme behind the strategy of Boyer and Moore is that only recursion variables are suitable candidates as induction variables, since only these will allow the induction conclusion to be evaluated. There are some refinements to this basic approach on which I will come back.

For the time being, I will further constrain the fundamental idea of Boyer and Moore by focusing on certain recursion terms of particular importance. With the help of this original tool, we can give a fresh look at the selection of induction variables, and especially link it with generalization.

If we allowed ourselves to talk of (symbolic) evaluation as regards the application of k-recursive definitions, we might as well also talk of computation rule. A computation rule tells us which subterm of a term to apply the k-recursive definition rule to. However, our weak completeness theorem of chapter 1 holds independently of any computation rule; that is, no matter the order in which are evaluated the subterms of a boolean term without vacuously defined function constants or variables, it will always be reduced to true or false. This is also consistent with the totality of our definitions. If nothing can be gained from a completeness point of view by introducing the notion of computation rule, we can

however make it to profit as regards efficiency. If our analogy between recursion and induction stretches far enough, we can say that selecting induction variables with the help of an efficient computation rule for k-recursive definitions will lead to a more efficient induction tactic, that is, one which will yield subgoals easier to solve.

An optimal computation rule is known for recursion equations. It bears the name of normal rule in Manna, Ness and Vuillemin (1971), of delay-rule in Vuillemin (1973), and is more widely known as call-by-need. An adaptation of Vuillemin's idea is used in the implementation of an efficient simplification algorithm which will be discussed in chapter 6. For the moment, I will focus on how the call-by-need concept can be helpful in selecting induction variables. The starting point is again quite simple. What do we need to know about a function application in order to be able to apply the k-recursive definition rule to it? We need to know the values of its recursion terms, if it has any. In effect, definitions are applied by matching; the non-recursion terms will always match the parameters of the definitions, but the recursion terms must at least have a constructor constant as rator in order to match their counterparts in the definitions. So, they need to be evaluated before anything else. (Vuillemin's recursion equations are a bit different since the left of an equation is a function application whose arguments are all parameters, i.e. variables, so that they always match any term.) Once the definition has been applied, we can use Vuillemin's technique for keeping track of sharing. The interesting point is that if we apply the call-by-need line of reasoning to an induction goal which has already been simplified, the

process will be stopped by one or more variables marking argument positions which the call-by-need evaluator must have more information about: I submit that these variable occurrences constitute excellent candidates for doing induction upon. I call them primary variable occurrences.

The efficiency of this approach can be justified thus. Any non-primary variable occurrence, even a recursion one, lies in some non-recursion term; the primary variable occurrences are the only ones to be only in recursion terms, if we consider the whole goal as a recursion term. So, if we replace the primary variable occurrences by structures as in the induction conclusion, and if we simplify the resulting term, the effect of evaluating the function applications containing the structures as recursion arguments may ripple up and trigger the evaluation of the whole goal, thus allowing a better possibility of matching with the hypotheses. The similar effect of evaluating a function application for which a non-primary variable occurrence has been replaced by a structure cannot ripple up to the top since it will be stopped at the level of the non-recursion term in which a non-primary occurrence necessarily appears.

A simple example can helpfully illustrate this. Take the goal

app(app(j,k),l)=app(j,app(k,l)).

We start the chain of reasoning with the function constant =; both of its arguments are recursion arguments. So, we need to evaluate both of them before being able to apply the definition of =. We iterate the process: to know about app(app(j,k),l), we must know about app(j,k), and to know about app(j,k), we must know about j. But we know nothing about j; so, this primary occurrence of j makes a good

induction candidate. On the right of the equality, to evaluate app(j,app(k,l)), we must know about j again. So, this variable is undoubtedly the induction variable to choose according to this technique (and as a matter of fact, a mechanical proof is obtained in a single induction on (j)). Note its directedness: k and l are never considered. Now, if we replace j by cons(n,j) as in the induction conclusion, the left becomes app(app(cons(n,j),k),l); its evaluation successively gives app(cons(n,app(j,k)),l) and cons(n,app(app(j,k),l)). The right becomes app(cons(n,j),app(k,l)) and its evaluation is cons(n,app(j,app(k,l))). Finally, the definition of equality becomes applicable. This is what was meant by evaluation rippling up to the top. This is only possible with j; k has a non-primary recursion occurrence in app(j,app(k,l)), but evaluation cannot go beyond the non-recursion term app(k,l). This gives only the rudiments of a method which will be refined in section 4 of this chapter. The interesting fact about this approach to induction variable selection is that generalization can be integrated to it in a natural way. We must go back to our analogy between recursion and induction. I have said that generalization followed by induction on the resulting variables can be viewed as induction on certain occurrences of terms. Which term occurrences in the goal can we consider as better candidates for induction than the primary variables? Simple answer: the term occurrences leading to them by the call-by-need evaluation, or in other words, the term occurrences in which the primary variable occurrences appear. I call these primary term occurrences, including the primary variable occurrences. Generalizing these term occurrences can be seen as a short cut toward the evaluation of the whole induction goal; and again, if our

analogy is good enough, we can suppose that such generalizations will lead to shortened proofs by induction.

Here is an example bearing the same flavour as the previous one

app(app(rev(j),k),l)=app(rev(j),app(k,l)).

We do as before except that for each term occurrence considered by the call-by-need evaluator, we ask the question: can this occurrence (maybe together with others) be generalized? The answers given by our checker are negative, until we get to rev(j): if we replace both occurrences of it by a variable, the new subgoal is still provable (it is actually the same as the previous example). The new variable is chosen to be the induction variable. The advantages of this purposeful generalization is that we can meaningfully generalize only certain occurrences of a term and in particular of a variable. For example, with

app(app(j,j),j)=app(j,app(j,j)),

we find that the first and the fourth occurrences of j are primary occurrences. We try to generalize them to a new variable which successfully yields

app(app(k,j),j)=app(k,app(j,j)).

This subgoal can be proved with one induction on (k).

Note two points: (1) non-provability checking is essential to such a generalization method, and (2) the approach of Boyer and Moore, and of Brotz to generalization as separated from induction variable selection leads in this example to the non-provable subgoal app(k,j)=app(j,k). A refined generalization tactic is given in the following section.

In conclusion, by pushing the analogy between recursion and induction with the help of the call-by-need notion, we obtain a heuristic integration of generalization and selection of induction variables which we have shown to be formally related in the previous section.

3.3 GENERALIZATION

For ease of implementation, the algorithms for generalization and selection of induction variables are actually separated but make use of the same principles. A first version of the generalization method was given in Aubin (1975). However, before giving the precise algorithm, I will try to clarify some points which have been skimmed over in the previous section.

The first question has to do with the effect of simplification rules over the present frame of work. Indeed, we have implicitly assumed so far that no such rules are used in addition to k-recursive definitions. Take the following definition of +:

```
m+n : nat <=
cases m [zero <= n |
        succ(m) <= succ(m+n)];
```

it is recursive on the first argument. So, only variables occurring in the corresponding position can be candidates for induction. But, if we have proved and added as theorems the following equalities: m+zero=n and m+succ(n)=succ(m+n), then the recursion and non-recursion arguments of + become equally good choices for generalization and induction. In effect, either will permit an induction conclusion to simplify to a near match with its induction hypothesis. Our strict call-by-need approach asks for revision if we

want the induction tactic to take into account the presence of simplification rules. This appears to be a complex question, and I have stopped short of tackling it.

In practice, this prover does not start from definitions only, but has available to it all sorts of simplification and normalization rules about connectives employed in putting terms into normal form. So, in our algorithms, we can at least take this special knowledge into account: for =>, the recursion arguments will be considered to be all antecedent and consequent members, and there will be no non-recursion arguments; for & and v, the recursion arguments will be all the conjuncts and disjuncts respectively, and there will be no non-recursion arguments. Simplification rules about other function constants will not be taken into account.

The second question is more directly pertaining to generalization. As seen in the previous section, when a call-by-need evaluator tries to reduce a simplified induction subgoal, it successively encounters term occurrences which would make good induction candidates if they could be generalized to variables. However, if we do this for every occurrence and check the generalized terms for provability, the rate of success is very low. In other words, most attempted generalizations will be unsuccessful. In chapter 2, we rejected the use of such tactics with low levels of productivity. So, we must tighten the context in which generalization is used; we must find a set of occurrences of a term, including the primary occurrence at stake, such that the generalization tactic has now a high probability of success.

For this purpose, I have adapted an idea of Boyer and Moore: if a term has primary occurrences in both arguments for the function constants => and =, we first try to generalize these occurrences to a new variable; if this fails, we try to generalize them together with non-primary occurrences. (I suspect that the reason why this is helpful is because of the reflexivity property of => and =; if this is the case, any reflexive relation would do. On the other side, irreflexive relations may also be helpful.) This is a compromise. On the one hand, it would sometimes be useful to generalize only one primary occurrence of a term. For example, this goal

```
ord(app(rev(l),k))
=>
ord(k)
```

could usefully be generalized to

```
ord(app(j,k))
=>
ord(k),
```

if only the generalization of the primary term rev(l) to the variable j were allowed. On the other hand, generalization is not always safe and requires checking. In his thesis (1973), Moore gives the example of

```
sort(sort(l)) = sort(l)
```

for which sort(l) cannot be replaced by a new variable without loosing provability.

I can now present the algorithms of the generalization tactic.

Generalization of induction goal

We start with two null lists: fail, which will contain the subterms of the goal for which generalization will have failed; and succ, which will contain the occurrences of the subterms of the goal for which generalization will have succeeded. The induction goal is assigned to a program variable t.

If t is a variable, we terminate. Otherwise, if its rator is a constructor constant or a vacuously defined function constant, we recursively apply this part of the algorithm to its arguments. In all other cases, for each recursion argument s of the function application, if s is a member of fail or if some member of succ appears in s, we recursively apply this part of the algorithm to s. Otherwise, we try to generalize s in the induction goal (see next algorithm): if this succeeds, then we have a new induction goal and we add all the generalized occurrences of s to the list succ; if this fails, we add s to the list fail and we recursively apply this part of the algorithm to s. We terminate when all recursion arguments have been accounted for. ||

Generalization of a term s in a term t

If t is a variable, we terminate. Otherwise, if it has => or = as rator, we look for a primary occurrence of s on each side of the function constant. If there is one side for which we cannot find such an occurrence, we terminate. Otherwise, we successively replace by a new variable the primary occurrences of s, then these occurrences together with a subset of the non-primary occurrences of s (preference is given to recursion occurrences), until either we find a generalization of t which is checked provable, in which case

we terminate the search with the generalized term, or we exhaust all possibilities, in which case we recursively apply the algorithm to the recursion arguments in t. ||

I have not explored in detail good tie-breaking rules for this strategy; for the moment, the tactic tries to generalize the smallest number of primary occurrences.

Here are some examples.

Example 1

The original goal is the associativity of * (multiplication): (m*n)*p=m*(n*p). After one induction on m, we obtain the subgoal

(n+(m[1]*n))*p=(n*p)+((m[1]*n)*p).

Both * and + are recursively defined on their first arguments; so, the primary terms on the left of the equality are: the whole left-hand side, n+(m[1]*n), and n (first occurrence); and on the right: the whole right-hand side, n*p, and n (third occurrence). Among all these, the primary occurrences of n are the only ones which can be generalized, which yields:

(n[1]+(m[1]*n))*p=(n[1]*p)+((m[1]*n)*p).

This generalized subgoal is solved in one induction on n[1], plus another generalization-induction step. ||

Boyer and Moore, on the other hand, would generalize this goal to

(n+n[1])*p=(n*p)+(n[1]*p),

because m[1]*n occurs on both sides of the equality. Their generalization tactic is blind to whether a term occurs in a primary recursion position or not: this concept does not exist as far as they are concerned. We have said that only the generalization of

primary term occurrences is directly useful to induction; yet, the Boyer-Moore generalization above is useful. By generalizing m[1]*n, they luckily eliminate both non-primary occurrences of the variable n, leaving the way clear for inducing only on the primary occurrences of this variable. Our tactic achieved the same result in a more purposeful way. Even if the Boyer-Moore tactic is successful for the wrong reason, one cannot help feeling that the resulting generalized term is better. It is better because it is simpler. Indeed, one should aim at keeping subgoals simple: they are then easier to read, easier to check, etc. Now that I have made my point about primary occurrences of terms, one could first try to generalize terms in any positions, so to speak, blindly, and then only use the careful mode of generalization. However, one must not forget that generalization is costly because of the checking involved and finding a generalization to a term which turns out to be useless for induction is just a waste of resources. So, in the end, it is a question of trade-off, but as far as I am concerned, I will make use of the purposeful generalization tactic only.

Example 2

This is a case where our generalization tactic must be used:

n*(n+n)=(n*n)+(n*n).

The first and fourth occurrences of n are primary. This time, we also have other recursion occurrences: the second and the sixth. The tactic first tries to generalize only the primary occurrences which fails. It then tries to add some non-primary occurrences to the set of occurrences to generalize; it succeeds after a few attempts with:

n[1]*(n+n)=(n[1]*n)+(n[1]*n).

This subgoal is proved with four inductions, three of which require generalizations of the same kind; the whole proof is displayed in section 1 of appendix 3. Note again that this problem could not be solved without the help of our checker which rejects non-provable trials. ||

Example 3

The original goal is

subset(k,k).

No generalization is possible and induction is done on k, although some trouble could be foreseen because of the non-primary occurrence of k. Indeed, we obtain this subgoal for which the induction hypothesis cannot be used:

```
subset(k[1],k[1])
=>
subset(k[1],cons(n[1],k[1])).
```

The first and third occurrences of k[1] are primary and can now be generalized to yield the subgoal:

```
subset(k[2],k[1])
=>
subset(k[2],cons(n[1],k[1]))
```

which is easily proved in one induction on k[2]. ||

3.4 SELECTION OF INDUCTION VARIABLES

Now that all useful generalizations have been done, we are left with selecting a list of induction variables. We have seen that these two strategies are two faces of the same coin; they have been separated in practice only for convenience. However, the underlying

principles behind them remain the same, and the selected variables will be variables resulting from generalization, unless some other variables are just as good. There are in fact two aspects to the selection: (1) the proposal of a certain number of candidate lists of variables, and (2) the election of a unique candidate, possibly after some coalitions. Before giving the algorithms, I would like to point out a few refinements which should be brought to the basic primary variable occurrence idea. Most of them have actually been adapted from Boyer and Moore.

Firstly, if all recursion arguments in a primary function application are variables, we take them all as candidate list. In effect, if the induction conclusion is to be evaluated for this application, then all recursion variables must have been replaced by a structure; otherwise, definitions could not be applied which would spoiled the effect wished for. There is an exception to this rule which is original to this prover: applications with constructor constants as rators may appear as recursion arguments of a primary application together with variables; they are ignored for the purpose of proposing a candidate. On the other side, if one or more recursion arguments of the primary application are function applications (with defined function constants as rators), then we must look at the primary terms of these applications for candidates. They are all equivalent from our point of view, so each one will propose one (or more) candidate list.

The second point, which has the same rationale behind, is about unioning all candidate lists which share a variable. This is indeed in agreement with the idea that if an application has only variables as recursion terms, all or none of them should be proposed

as candidates.

The third practical aspect I would like to put in evidence is original: it concerns non-primary occurrences of a variable. The selection is further constrained by encouraging variables with many primary occurrences and few non-primary occurrences. The reason for doing this can be understood if we go back to our chief preoccupation of generating induction subgoals which are easy to solve, that is, whose induction hypotheses and simplified conclusions match as closely as possible. We pick variables in order to induce on them. But, if a function application has an induction variable in a non-primary position, we have seen that it will not in general be possible to simplify it very far once the variable is replaced by a structure; this would prevent an optimal matching of the hypotheses and conclusions.

Fourthly, we single out the candidates whose variables do not occur as accumulators in the goal (see chapter 4). We will see that such variables need often to be instantiated in the induction hypotheses, which cannot be done if they are induction variables as well.

If we still have a tie at this point, we ask the question: how far up is the evaluation of the induction conclusions going to ripple for each candidate? We have seen that the further it ripples, the better will the match be with the hypotheses. This allows us to indirectly take into account the definitions of the function constants in the primary applications.

Finally, we prefer variables which have never been used for induction on a given branch of the proof tree. This acknowledges the possibility of making a bad choice at some point and allows for a partial recovery of such a mistake.

Proposal of candidate lists of variables

We start with a term t and a null list of candidate lists.

If t is a variable, we terminate. Otherwise, if its rator is a constructor or vacuously defined function constant, we recursively apply this part of the algorithm to its arguments. In all other cases, if all recursion arguments are either variables or have a constructor constant as rator, we constitute the list of all the variables into a candidate list; otherwise, we recursively apply this part of algorithm to each recursion argument.

We finally terminate with a list of candidate lists. ||

Election of a unique candidate list

We start with a list of candidate lists. If there is only one candidate, we terminate.

Otherwise, we union all lists having at least one variable in common. If only one list remains, we terminate.

Otherwise, we compute the total number of non-primary occurrences for the variables in each candidate list; we retain only the candidates marking the lowest. If there is only one remaining candidate, we terminate.

Otherwise, if the variables in the remaining candidates have no non-primary occurrence, we union them all and terminate with this candidate.

Otherwise, we eliminate the candidates containing variables which also occur as accumulators. If only one candidate remains, we terminate; but if no candidates are left, we recover the eliminated ones.

Otherwise, for each candidate, we apply to the induction goal the conclusion substitutions corresponding to non-empty hypothesis substitutions (see chapter 5); we count how many times the k-recursive definition rule can be applied to the primary applications in all, and divide the result by the number of substitutions. We retain the candidates which score the highest. If only one candidate remains, we select it.

Otherwise, we try to find a candidate list whose variables have never been used for induction on the current branch of the proof tree. If we cannot find any, we pick the first candidate of the list. ||

Brotz's method for selecting induction variables (1974) can be explained in our theory. His functions are all binary, and the recursion argument is always the second one. In order to choose an induction variable, he selects the rightmost variable on each side of equality. But in our terms, these happen to be primary variables. If they are the same, then it is undoubtedly the induction variable to select; otherwise, Brotz chooses at random, which is a poor tie-breaking rule.

<u>Example 1</u>

The goal

m=<n & n=<p => m=<p

yields the candidate lists (m,n), (n,p), and (m,p). They are all

unioned, since they share a variable two by two. So the unique final list is (m,n,p). The induction tactic then yields eight subgoals which are all immediately simplified to true. ||

Example 2

We have

app(j,k)=app(j,l) => k=l

to prove. Our candidate lists are (j), (j), and (k,l). After unioning, we are left with (j) and (k,l). But, k and l have both non-primary occurrences. So, we retain (j) only. Inducing on (j) is sufficient to prove the goal. ||

Example 3

This again is a simple example. We must prove

app(rev(l),rev(k))=rev(app(k,l)).

We have two candidates (k) and (l). Both variables have one non-primary occurrence; so, we have to look at the behaviour of the induction conclusions for each candidate. If we replace k by cons(n[l],k[l]) in the goal, we cannot evaluate any primary applications on the left-hand side of the goal; in effect, we need not look at rev(k) since it is not a primary application. On the right-hand side, we can evaluate

app(cons(n[l],k[l]),l)

to

cons(n[l],app(k[l],l)),

and then

rev(cons(n[l],app(k[l],l)))

to

app(rev(app(k[l],l)),cons(n[l],nil)).

So, (k) scores 2. With the second candidate, we replace l by cons(n[l],l[l]): only the definition of rev can be applied on the left of the goal and no definitions at all on the right. So, (l) scores 1. Consequently, we choose (k). This choice is quite important: since with (k) the effect of evaluating the conclusion can propagate higher up, replacement is made possible with the hypothesis (see chapter 5), whereas this is not the case with (l). The goal can then be proved in two generalization-induction steps, in addition to the induction on k. ||

3.5 GENERALIZATION AND STRENGTHENING

This short section is prospective. It discusses a form of generalization which would very much help to keep subgoals simple.

In the previous sections, we have limited the generalization tactic to operate on terms which occur on both sides of = or =>. However, we made the point that this was only a compromise, since it would sometimes be useful to generalize only a single occurrence of a term. The question I want to address is: what does it mean to generalize an antecedent or consequent member of an implication? We have seen that these can be considered as primary term occurrences.

Suppose we have the goal s=>t. If we generalize s to variable x, we get x=>t. Induction on x yields the subgoals true=>t and false=>t; the first one simplifies to t and the second one, to true. An analogous phenomenon occurs if we generalize t. But this has already been seen in the derivation of the weakening rule: generalizing an antecedent or consequent member is equivalent to removing it from the antecedent or consequent. I call this

strengthening; it is just a particular form of generalization followed by an induction.

How is strengthening useful? We will see in chapter 5 that it is used to reject useless induction hypotheses and to discard useful hypotheses once they have been made to profit. It would be very nice if one could go further than this, and discard from a goal <u>any</u> antecedent and consequent member thought to be useless. Of course, as for the case of generalization, this cannot be used safely without checking.

For instance, this subgoal came up in my (vain) attempts at proving a tree sort algorithm:

```
ord(app(flattree(totree(n,t[1])),flattree(t[2])))
& n[2]=<n
=>
ord(flattree(totree(n,t[1])))
v ord(flattree(totree(n,t[2])))
v ord(app(flattree(t[1]),flattree(t[2])))
```

It would nice to be able to reduce it to, say:

```
ord(app(flattree(totree(n,t[1])),flattree(t[2])))
=>
ord(flattree(totree(n,t[1])))
```

by removing some antecedent and consequent members.

The problem with such a tactic concerns the relevance of an antecedent or consequent member not so much from a consistency point of view (this can be checked), but from a completeness one. For example, we will see that our induction tactic can reject an induction hypothesis on some firm evidence that it will not be useful to the proof. The question I am asking is the following : can we find some heuristic criteria regarding the completeness of a general strengthening tactic, that is, criteria which would tell that a subgoal can be proved more easily if we discard such and such

antecedent and consequent members? Hoping for a general answer to this question would be very optimistic. For one thing, deleting a consequent member is like crossing out a possible or-choice; as we have seen in the previous chapter, making or-choices is a delicate matter.

Certainly the blind elimination of antecedent or consequent members cannot be a good strategy in general, even when the result cannot be checked non-provable. In particular, one should never discard an induction hypothesis at this stage: if it has not already been discarded in the induction or replacement tactics, then it should be still be necessary to the proof.

CHAPTER 4

INDIRECT GENERALIZATION

Now that we have a method of selecting induction variables (which may involve generalizations), we are almost ready to explain how induction subgoals can be generated on the basis of these variables. But beforehand, a problem has to be discussed whose solution also leads to some kind of generalization. In the foregoing chapter, we focussed our attention on recursion arguments, and in particular, primary ones. In this chapter, we will give a closer look at non-recursion arguments which do not stay fixed in k-recursive definitions. The first section of this chapter is introductory; the second section explains the technique of indirect generalization; finally, the third section discusses alternative solutions.

4.1 PRELIMINARIES

I will introduce the problem with an example. Take the following definitions for a function which reverses a list:

```
rev2(l) : list <= rev2a(l,nil)

rev2a(l,k) : list <=
cases l [nil <= k |
        cons(n,l) <= rev2a(l,cons(n,k))]
```

The first and second arguments in rev2a(l,k) are recursion and

non-recursion arguments respectively. However, by inspecting the definition of rev2a, we find that the non-recursion argument k does not stay fixed on the right of the definition but becomes cons(n,k). This is quite legitimate according to our scheme of definition by k-recursion. The interest of such definitions lies in the fact that for the class of problems studied, they are literal translations of iterative programs. For example, the definition of rev2 comes from the program:

```
begin
y[1], y[2] := 1, nil
while y[1] /= nil
    do y[1], y[2] := tl(y[1]), cons(hd(y[1]),y[2]) od
k := y[2]
end
```

Such non-fixed non-recursion arguments are called <u>accumulators</u> (following Moore 1974), since they can be considered as holding current values of computations. A term occurring in an accumulator argument position is also called an accumulator.

Perhaps one can already see the difficulty involved in proving properties of functions like rev2a by induction: a nice match between induction hypotheses and evaluated conclusions is less likely to be reached. In effect, suppose that non-recursion arguments do stay fixed, then by applying k-recursive definitions to induction conclusions, we can often cause the same function applications to occur in both the hypotheses and the conclusions since definitions are recursive. This will be explained and illustrated in detail in the next chapter. Suffice to say for the moment that applying definitions with accumulators will not cause the recurrence of identical function applications since this one-step evaluation will have changed the accumulator.

This last point should be exemplified. Suppose that we want to prove

```
rev2a(k,l)=app(rev(k),l).
```

By our technique, we choose to induce on (k). Consider the induction subgoal for which k is replaced by cons(n[l],k[l]) in the conclusion:

```
rev2a(k[l],l) = app(rev(k[l]),l)
=> rev2a(cons(n[l],k[l]),l) = app(rev(cons(n[l],k[l])),l).
```

This is simplified to:

```
rev2a(k[l],l) = app(rev(k[l]),l)
=> rev2a(k[l],cons(n[l],l))
   = app(app(rev(k[l]),cons(n[l],nil)),l).
```

The hypothesis differs from the evaluated conclusion; we could have hoped that at least one side of the equality would match, but this is not even the case. However, this undesirable situation could have been avoided, had we only been a little bit more careful than above. According to our induction rule, we had in fact the choice of replacing l by any term in the hypothesis; if we had looked ahead at the induction step, we would have found that l could have been instantiated to cons(n[l],l) with profit. If we take this course of action, we get the following (evaluated) induction subgoal:

```
rev2a(k[l],cons(n[l],l)) = app(rev(k[l]),cons(n[l],l))
=> rev2a(k[l],cons(n[l],l))
   = app(app(rev(k[l]),cons(n[l],nil)),l).
```

We now have a partial match and we can replace the left-hand side of the conclusion by the right-hand side of the hypothesis in the conclusion. The resulting subgoal can be proved easily. So, when accumulators are variables, we need not worry too much since they can be instantiated to provoke a good match. How this is performed will be explained in detail in the next chapter.

For the moment, I will focus on the case where accumulators are not variables. Suppose that we now have to show:

```
rev2a(k,nil) = rev(k).
```

Clearly, we must do induction on (k). The induction subgoal with the non-empty hypothesis is:

```
rev2a(k[l],nil) = rev(k[l])
=> rev2a(cons(n[l],k[l])),nil) = rev(cons(n[l],k[l]),
```

which is simplified to:

```
rev2a(k[l],nil) = rev(k[l])
=> rev2a(k[l],cons(n[l],nil)) = app(rev(k[l]),cons(n[l],nil)).
```

While in the previous example, we had the variable l in the second argument position of rev2a, which could be instantiated, we now have the constant nil. No replacement is then possible on the left of the conclusion. (We could replace rev(k[l]) by rev2a(k[l],nil) on the right of the conclusion, but that would simply put the matter off: the next subgoal would contain two non-variable accumulators.) Consequently, if we want to do as before, we must find a way of generalizing nil in rev2a(k,nil)=rev(k) to a new variable; this variable could then be suitably instantiated and make the replacement possible.

The following section will expound the method used by this prover for doing such generalizations. At present, I wish to make precise the conditions under which they will be attempted.

I have to introduce the notion of <u>induction applications</u>. An induction application is a primary function application of the induction goal, each recursion argument of which either (1) is an induction variable, or (2) has a constructor constant as rator, or (3) is an induction application itself. In other words, they are the class of primary applications in which the induction variables occur

as primary variables. The definitions of the leading rators in these applications stand a chance of being used in the evaluation of the induction conclusion; that is why they are important. For instance, in the simple examples above, all function applications of the induction goal were induction applications; however, we did not go as far as using the definition of = in either case. Such applications will also play a useful role in the generation of induction subgoals to be studied in the next chapter.

With this notion, I can now give the simple principle which guides the decision when and what to generalize: we try to generalize those non-variable accumulators which occur as arguments of induction applications, since only these may prevent a good match between hypotheses and conclusions. In the last example, the induction application rev2a(k,nil) has the non-variable accumulator nil which, according to this rule, we should try to generalize.

To conclude this section, I would like to point at two differences between this generalization problem and the one studied in the previous chapter. The first discrepancy regards the strategy used. In chapter 3, we limited ourselves to replacing a term by a variable only if it had a primary occurrence in both arguments of an equality or implication. We made the remark that this was not a question of principle but efficiency. If that condition was not met, or if the generalized subgoal could be checked non-provable, we just carried on with induction without generalizing. In this chapter, we want to be more persistent in our trials and force generalization upon induction goals whenever possible. For instance, nil in rev2a(k,nil)=rev(k) does not have an occurrence on both sides of =. Nevertheless, we will try to generalize nil, and it is because of

this intention that the present generalization method has to be quite elaborate.

The second dissimilarity between this and the previous type of generalization is more of a theoretical interest. In the preceding chapter, we deduced from Kreisel and Prawitz's results that any complete backward search strategy had to involve generalization, at least for the arithmetic subset of our system. We can now ask whether accumulator generalization is also inherent to inductive proofs. This time, we get our answer from Peter (1967). We can read her results as follows: at least in the context of arithmetic, a definition by k-recursion involving accumulators can be reduced to a definition of the same k-recursive function without accumulators. So, if accumulators can be eliminated from our function definitions, they can also be eliminated from our inductive proofs. Hence, accumulator generalization can theoretically be waived by using this reducibility result. But in chapter 1, we decided to deal with functions as they were written and not to take possible reductions into account. I will follow up this line. This is debatable, and some people (Burstall 1974) have argued in favour of systematically doing the transformation to accumulator-free definitions. Moore (1974), who also tackles this problem, appears to have a mixed attitude on the question. The third section will discuss this alternative line of attack in more detail. For the moment, the generalization problem remains for us and a solution to it is given in the following section.

4.2 INDIRECT GENERALIZATION

I must start this section with a word of caution. Whereas most of the other algorithms presented in this dissertation achieve a great degree of generality, the procedures to be described in this section have a more restricted range of application. They work within induction on one variable leading to one basis and one induction step in the traditional sense; moreover, the latter can have only one hypothesis. In practice, this means induction on one variable of type nat or list. Furthermore, definitions can have only one accumulator. How these restrictions can be lifted is still an open question.

As mentioned, we decide about accumulator generalization in an induction goal after a few observations: we first select the induction variable and find the induction applications. We check whether any of these have non-variable accumulators and only then is the present tactic applied.

We can go on from this point with our example. We know that we must generalize nil in rev2a(k,nil)=rev(k). However, we do not have an occurrence of nil on both sides of =, and we know that generalizing in such a context is not very productive in general. As a matter of fact, the generalized term rev2a(k,l)=rev(k) can be shown to be non-provable by our checker. Instead of abandoning our trial, we ask the question: how can we massage our induction goal so as to make nil recur on the right of the equality? Answering such a question is what our method does. Intuitively, if we know that app(l,nil)=l, we can rewrite rev(k) as app(rev(k),nil). So, our goal becomes rev2a(k,nil) = app(rev(k),nil), and nil occurs on both sides.

What if we replace it by a new variable? We get rev2a(k,l) = app(rev(k),l) which is provable; this subgoal is actually the first example given in the previous section.

Such generalizations can be found systematically. There are two parts to the method: (1) we find for which subterms, say s[1] and s[2], of the induction goal the hypothesis fails to match the evaluated conclusion (at least one of s[1] or s[2] is a non-variable accumulator, say it is s[1]); (2) by specialization of the induction goal, we try to express s[2] in terms of s[1], say s[2]=w[s[1]], and then by replacing s[2] by w[s[1]] in the goal, we are left with a term having two occurrences of s[1] (the non-variable accumulator) which can then be hopefully generalized to a new variable.

The following two subsections explore and illustrate these techniques in detail. A first version of them is given is Aubin (1975).

4.2.1 Search For Mismatches

We want to find for which terms, in addition to the non-variable accumulators, hypotheses and conclusions fail to match. It is believed that such terms and the non-variable accumulators will in general stand in some meaningful relation, each on one side of = or =>. Which relation this is precisely will be found by the second part of the method. We will now see how mismatches are found.

What can we learn from the prospective induction step of our example, namely,

```
rev2a(k[1],nil) = rev(k[1])
=> rev2a(cons(n[1],k[1]),nil) = rev(cons(n[1],k[1])).
```

We evaluate the conclusion and compare the following two terms:

1. rev2a(k[1], nil) = rev(k[1])
2. rev2a(k[1],cons(n[1],nil)) = app(rev(k[1]),cons(n[1]),nil))

Expectedly, nil does not match cons(n[1],nil), because it is the accumulator at the origin of the problem. On the other hand, we have that rev(k[1]) fails to match app(rev(k[1]),cons(n[1],nil)). However, I am not happy with the latter because the induction variable k[1] occurs in the mismatched term rev(k[1]). In the next section, we will see that k[1] is to be specialized to nil. If we allow k[1] to occur in the mismatched term, this one will be modified by specialization and any later replacement will become impossible (this will be explained in detail later). Consequently, if a mismatched term contains the induction variable, the whole method fails; in fact, mismatched terms should contain as few variables as possible.

However, rev(k[1]) occurs in app(rev(k[1]),cons(n[1],nil)) and we are allowed to think that in such circumstances, the mismatch can be made more local. That we should strive for the most localized mismatches is a clear consequence of our objective of maximizing the match between hypotheses and conclusions. Non-variable accumulators are an exception to this minimization of mismatches since we want the whole of them to be eventually replaced by variables.

So, a mismatch between s and t is not acceptable if the induction variable occurs in s; but then, s occurs in t under normal circumstances. How can we make use of this observation in order to obtain a mismatch which does not involve the induction variable? In our example, we can look for an l such that rev(k)=app(rev(k),l); clearly, l should be nil. Now after replacing rev(k) by

app(rev(k),nil) in the goal, the mismatch on the right is between app(rev(k[l]),nil) and app(rev(k[l]), cons(n[l],nil)), which reduces to a mismatch between nil and cons(n[l],nil); this meets our conditions.

The process of finding a value for l and replacing the left by the right-hand side of the equality above is called expansion. In general, if s mismatches t and s occurs in t, (1) we replace in t the biggest subterms not containing s by new variables to get, say, w[s,x*], and (2) we look for instances u* of these new variables such that s=w[s,u*]. If it can be done, we replace s by w[s,u*] in the goal.

Life will not in general be so easy. Mismatching and expansion may be difficult to achieve and I would like to present two problematic situations for which I had to find special solutions.

The first one has to do with mismatches involving constructor constants e.g. s against succ(s), s against cons(u,s), where the induction variable occurs in s. This is a relatively frequent situation and there is no hope of being able to expand s, as things stand. In such cases, we rewrite e.g. succ(s) as succ(zero)+s, or cons(u,s) as app(cons(u,nil),s), before attempting to expand s which will then normally be successful. Clearly, the expansion of succ and cons can only be performed when + and app have been defined. We then have in these definitions all the information necessary to replace e.g. succ(s) by succ(zero)+s. But lacking of a uniform method for accessing it, I had to revert to this more special purpose method. Actually, Katz and Manna (1973) make use of the same pregiven information in their system for discovering inductive assertions. It is found in an operator table which gives the general computation of

e.g. succ after n iterations.

The second special situation is related to certain definitions using conditionals. Take for example, the definition of union2:

```
union2(k,l) : list <=
cases k [nil <= l |
        cons(n,k) <= cond(mem(n,l),union2(k,l),union2(k,cons(n,l)))
```

This is a natural way of defining it. But one can view the right of the second branch of the definition as originating from the term

```
union2(k,cond(mem(n,l),l,cons(n,l)))
```

and having been normalized later. (There is a normalization rule which pulls conditionals as far out as possible; see chapter 6.) In the above term, the accumulator

```
cond(mem(n,l),l,cons(n,l))
```

sticks out clearly, whereas it does not in the definition of union2. Consequently, in cases like union2, conditionals are pushed inside whenever they can be localized to an accumulator position; this is done in the expansion process.

The algorithms can now be given.

<u>Search for mismatches</u>

We start with term s (the induction hypothesis, which turns out to be identical to the induction goal because of the restrictions) and term t (the simplified induction conclusion); we also have the list a of non-variable accumulators occurring in the induction applications. We initialize to nil the list m which will contain the mismatched terms of s.

If s is a member of the list a, then it is an accumulator and we add it to the list of mismatches m, unless it is equal to t. Otherwise, if s is a variable and is not equal to t, we try to expand s and t (see next algorithm). If we are successful, we apply this part of the present algorithm to the expanded s and t; otherwise, we add s to the list m of mismatches. If t is a variable, we add s to m. Otherwise, both s and t are function applications. If their rators are equal, we recursively apply this part of the algorithm elementwise to their rands; we may obtain a new s in the process because of expansions. If their rators are not equal, then we try to expand s and t, and we either recursively apply this algorithm to the expanded s and t if successful, or we register s as a mismatch in m, otherwise.

We terminate with (1) possibly a new s because of expansions, and (2) the list m of mismatches in s.

Before giving examples of mismatching, I will immediately describe the algorithm for doing expansion. Recall that expansion is necessary if the mismatch between s and t is such that the induction variable occurs in s.

Expansion

We start with the mismatched terms s and t, and we either fail or return s and t, at least one of which having been expanded.

If the induction variable does not occur in s, or s does not occur in t, then expansion fails: there is no need for it. If t has the form succ(t), it is rewritten as succ(zero)+t; if it has the form cons(u,t), it is rewritten as app(cons(u,nil),t). If t satisfies the scheme cond(a,f(x),f(y)), where x and y are accumulators for some f,

it is rewritten as f(cond(a,x,y)) and we terminate immediately with s and this new t.

Otherwise, we replace in t the biggest subterms of t not containing s by new variables. Suppose we thus get w[x,s], where x is one of the new variables. We generate the first few structures c[1], c[2], c[3], etc. for the type of x (see section 2.5), and we successively replace x by c[i] in w[x,s] until w[c[i],s]=s can be simplified to true. Then we have found w[c[i],s] as an expansion of s, and we terminate with w[c[i],s] and t. If we cannot expand s after trying with the first few structures for the type of x, we fail. ||

Here are two examples.

<u>Example 1</u>

We want to show the associativity of times2:

```
times2(m,times2(n,p)) = times2(times2(m,n),p).
```

But times2 is defined in terms of times2a which has an accumulator as third argument. The goal is actually simplified to:

```
times2a(m,times2a(n,p,zero),zero)
= times2a(times2a(m,n,zero),p,zero).
```

We find that we must do induction on (m) and consequently, the induction applications are

1. times2a(m,times2a(n,p,zero),zero)
2. times2a(m,n,zero)
3. times2a(times2a(m,n,zero),zero)
4. The whole goal.

So, all occurrences of zero except the first one are interesting accumulators. We must now try to mismatch the induction hypothesis (identical to the goal above) with the evaluated conclusion for which m has been replaced by succ(m). On the left, we must compare:

```
1.  times2a(m,times2a(n,p,zero),          zero          )
2.  times2a(m,times2a(n,p,zero),times2a(n,p,zero)+zero)
```

So, the first mismatched term to be recorded is the second occurrence of zero.

On the right, we must compare:

```
1.  times2a(times2a(m,n, zero ),zero)
2.  times2a(times2a(m,n,n+zero),zero)
```

So, the third occurrence of zero in the goal is also a mismatched term. Note that no expansion has to take place since in both cases zero is an accumulator.

In other words, the underlined terms in the following are the terms which should give us the key to our generalization problem:

times2a(m,times2a(n,p,zero),<u>zero</u>)
= times2a(times2a(m,n,<u>zero</u>),p,zero).

The obvious thing to try then is to replace both underlined occurrences of zero by a new variable. But the resulting term can be shown to be non-provable, and we will have to revert to a more sophisticated generalization method.

<u>Example 2</u>

We have the goal

```
mem(n,inter2(k,l))
=> mem(n,k).
```

The function constant inter2 is defined in terms of inter2a whose definition includes an accumulator. The goal is simplified to:

```
mem(n,inter2a(k,l,nil))
=> mem(n,k).
```

Induction is clearly on (k). Because the accumulator in the induction application inter2a(k,l,nil) is nil and not a variable, we have to generalize. The induction variable k is replaced by cons(n[l],k) in the conclusion of the induction step which is then evaluated. We

must try to mismatch the hypothesis and this evaluated conclusion. On the left, we compare:

```
1.  mem(n,inter2a(k,l,nil))
2.  mem(n,cond(mem(n[l],l),inter2a(k,l,cons(n[l],nil)),
                            inter2a(k,l,nil)))
```

But this is a case where the conditional in 2 should be pushed inside in order to get a clear mismatch. We thus get:

```
1.  mem(n,inter2a(k,l,                nil                   )
2.  mem(n,inter2a(k,l,cond(mem(n[l],l),cons(n[l],nil),nil)))
```

Consequently, we find our first mismatched term: it is the non-variable nil (against the conditional term).

On the right, we must compare:

```
1.  mem(n,k)
2.  or(n=n[l],mem(n,k))
```

This does not satisfy our conditions for a good mismatch, since the induction variable occurs in 1. We must try to expand 1. The biggest subterm of 2 not containing l is n=n[l]; so, we must find b such that mem(n,k) = or(b,mem(n,k)). After at most two trials, we discover that b should be true. So, we perform the expansion, and the mismatch problem becomes:

```
1.  or( true ,mem(n,k))
2.  or(n=n[l],mem(n,k))
```

Clearly, true is the second mismatched subterm in the expanded goal (against n=n[l]). As a result, we obtain:

mem(n,inter2a(k,l,<u>nil</u>))
=> or(<u>true</u>,mem(n,k)).

The underlined terms are those which are known to cause a mismatch between hypothesis and conclusion. This time, they are not identical so that we cannot even be tempted to generalize them directly. ||

4.2.2 Specialization And Replacement

I recapitulate the background leading to this second part of the indirect generalization tactic. We started with one or more non-variable accumulators which we knew would not match in the induction step. We wanted to generalize them to variables but this was not directly possible. By the processes of mismatching and expansion, we were able to find other terms, typically on the other sides of = or =>, which also failed to match.

The motivation behind the techniques of specialization and replacement is that the mismatched terms occurring on both sided of = or =>, e.g. the underlined terms in the previous examples, ought to be in some relationship with each other. Specialization finds this relationship, and replacement makes use of it. Brotz (1974) also used a specialization strategy to generate useful lemmas, but with no intent of provoking an indirect generalization.

Since it is difficult for a computer program to deal with underlined subterms, we choose to replace them by function applications; moreover, we can then control simplification better in the process of specializing. The new function constants are defined such that the resulting term is equivalent to the original goal, but the definitions are never applied. For example, we had:

    rev2a(k,nil) = app(rev(k),nil).

We simply define:

    f[1]() : list <= nil

    f[2]() : list <= nil

and we rewrite our goal as:

    rev2a(k,f[1]) = app(rev(k),f[2]).

The underlined nil's have been replaced by the constants f[1] and f[2] which are equal to nil.

We know that f[1] stands for an accumulator. So, we want to express f[2] in terms of f[1]. To this end, we replace the induction variable k by nil which is the necessary substitution for the basis of the induction. Thus, we obtain:

rev2a(nil,f[1]) = app(rev(nil),f[2])

which is simplified to:

f[1]=f[2].

As mentionned, simplification stops short of applying the definitions of f[1] and f[2], because they are the terms we are interested in. So, we have discovered the relationship between f[1] and f[2]. The equality above is actually true.

One should recall at this point the third of Polya's methods of working from a goal given in section 2.3: solving a less ambitious problem does not give the solution to the original goal but can provide some help toward its solution. This is what this specialization technique does. What is sought for is a relationship between the mismatched terms of our goal. These have been replaced by new function applications and specialization is a good way of obtaining this relationship. But why should we choose to specialize precisely by means of the substitution for the induction basis instead of any other structure? This is justifiable by looking at how functions with accumulators are typically defined: their values for e.g. zero, nil, are precisely the accumulators. For example, we have rev2a(nil,l)=l. This is not surprising since the second argument of rev2a is viewed as holding the current value of the computation of the function and this computation terminates when the first argument

becomes nil. Consequently, since we are trying to know more about accumulators, this specialization is likely to bring these to the forefront.

Here is the specialization algorithm:

Specialization

We start with the expanded term s and a list m of mismatched terms in s. We first replace in s each element of m by a new function application whose arguments are precisely the variables occurring in the mismatched term.

We find the induction variable of s (the first time round, this is given since s is then the original goal), we replace it by the basis structure for the type of this induction variable (typically zero or nil), and we simplify the result, without applying the definitions of the new function constants. If we cannot find an induction variable, we try with all the variables in s which do not occur in any new applications. If a new application occurs as the whole right or left-hand side of = or => in the specialized term, we terminate with this term. Otherwise, we apply this part of the algorithm recursively until we either succeed in finding a specialization or fail to find an induction variable; the whole of specialization then fails. ||

We can now see what becomes of examples 1 and 2.

Example 1

We start with

```
times2a(m,times2a(n,p,zero),f[1])
= times2a(times2a(m,n,f[2]),p,zero),
```

where the earlier mismatched terms have been replaced f[1] and f[2].

If we replace the induction variable m by zero and simplify the result, we get:

```
f[1] = times2a(f[2],p,zero).
```

This term is true and we have found the relation looked for. ||

Example 2

Our goal is rewritten as:

```
mem(n,times2a(k,1,f([1]))
=> or(f[2],mem(n,k)),
```

where

```
f[1]() : list <= nil

f[2]() : bool <= true.
```

When we replace k by nil and simplify the result, we obtain:

```
mem(n,f[1]) => f[2].
```

This is a suitable relationship between f[1] and f[2]. ||

Now, how can the specialized terms be put to use? One must have already guessed the answer. We use them to replace e.g. f[1] by f[2] or vice versa in the original goal, so that we have either all f[1]'s or all f[2]'s; then hopefully, direct generalization will be possible.

With our simple example, we had found the specialization

```
f[1]=f[2].
```

We remember that f[1] stands for the non-variable accumulator nil; so we replace f[2] by f[1] in

```
rev2a(k,f[1]) = app(rev(k),f[2])
```

to get

```
rev2a(k,f[1]) = app(rev(k),f[1]).
```

This can now be directly generalized to

```
rev2a(k,l) = app(rev(k),l).
```

How to do replacement with equality is quite clear. It is based on the first form of substitutivity given in section 2.4 (this rule is also used by the simplifier). Since our specialized terms are valid, using them in substitutions preserves the equivalence of the goal. In other words, only the last step of generalizing f[1] yielded a result which was not equivalent with the original goal.

However, we have to use a different rule with the second example above: we do not have an equality but an implication. So, we have to use the extended notion of substitutivity introduced in section 2.4 and apply it to =>. This was already discussed a little bit in chapter 2. It is worth giving the algorithm which does it. But before, we have to relax the rules given in chapter 2. We had:

```
u=>s     s=>t              t=>u     s=>t
-------------              -------------
    u=>t                        s=>u
```

We can relaxed them to:

```
u => s[1] v ... v s[n] v w'      s[1] & ... s[n] => t
-----------------------------------------------------
                      u => w
```

where each disjunct of t occurs as a disjunct of w, and w' is obtained by removing the disjuncts of t from w, and:

```
t[1] & ... & t[n] & w' => u      s => t[1] v ... v t[n]
-------------------------------------------------------
                      w => u
```

where each conjunct of s occurs as a conjunct of w, and w' is obtained by removing the conjuncts of s from w.

I will only give the tactic corresponding to the first rule.

<u>Replacement of consequent by antecedent</u>

We start with a goal w and an implication s=>t.

If w is not a conjunction or an implication, we fail. Otherwise, if w is a conjunction, we apply this part of the algorithm to each conjunct recursively. If w is an implication and if some disjunct of t is not a member of the consequent of w, we fail. Otherwise, we remove the disjuncts of t from the consequent of w, and we add each conjunct of s as a disjunct of the consequent of w. We then terminate with a new subgoal w. ||

There is an analogous tactic for replacement of antecedent by consequent, which corresponds to the second rule.

Contrary to replacement with equality, these tactics cannot be used safely without checking, since the rules above are not reversible. Nonetheless, we do not do any checking at this stage because generalization also asks for checking and we want to do it just once.

<u>Example 1</u>

By specialization, we have discovered the true fact:

```
f[1] = times2a(f[2],p,zero).
```

By replacement in the original goal, we obtain:

```
times2a(m,times2a(n,p,zero),times2a(f[2],p,zero))
= times2a(times2a(m,n,f[2]),p,zero).
```

Note that no generalization has been performed yet, that is, this term is still equivalent to our original goal. The whole process of finding mismatches, expanding, specializing, and replacing is a meaningful method of forcing generalization. In effect, we managed to provoke the recurrence of at least one of the original non-variable accumulators on both sides of =. We can now obtain by

direct generalization:

```
times2a(m,times2a(n,p,zero),times2a(o,p,zero))
= times2a(times2a(m,n,o),p,zero)).
```

It will now be possible to instantiate o (to n+o) and thus obtain a match for at least the right of the equality. The whole proof of the associative law of times2 is given in appendix 3, section 2. ||

Example 2

The goal was specialized to:

```
mem(n,f[1]) => f[2].
```

By replacing the consequent by the antecedent of this implication in the consequent of the goal (using the algorithm given above), we get:

```
mem(n,inter2a(k,l,f[l])
=> or(mem(n,f[l]),mem(n,k)).
```

In general, such a subgoal will not be equivalent to the original goal because of the non-reversibility of implication replacement; but as it turns out in this case, we get an equivalent subgoal. Generalization finally yields:

```
mem(n,inter2a(k,l,j))
=> or(mem(n,j),mem(n,k))
```

which can be proved in one induction. ||

4.3 DISCUSSION

The rationale behind the foregoing method was the following. We know that accumulators may be detrimental to a nice match between induction hypotheses and evaluated conclusions, since by definition, they are non-recursion terms which do not stay fixed. When accumulators in the induction applications of a goal are variables,

then an appropriate instantiation of the hypotheses will in general counteract the negative effect of the accumulator. This is not possible however when accumulators are not variables and the preceding technique was used to force the generalization of such accumulators to new variables, while retaining the provability of the goal.

Our technique for obtaining such a result does not have all the generality which one could have hoped for. However, it helps proving a reasonable number of interesting theorems as can be seen in appendix 2. It has two main advantages: (1) it yields natural generalized subgoals (this is important if our automatic methods are to be part of some man-machine system), and (2) when it is applicable, it does produce useful generalizations which allow the proofs to progress.

But as already mentionned, we know from Peter (1967) that at least for arithmetic, definitions with accumulators are reducible to definitions without such arguments and thus, the generalization problem can be averted. However, Peter's method involves decomposition into prime factors, and the resulting definitions are likely to look a bit artificial. If one was to extend Peter's result to our language, there is little doubt that the equivalent reducibility result would also be too general to be very practical.

Nonetheless, one can look for more useful, if less general, results in the same vein. Minsky (1967) shows the following result. Given function f defined thus:

```
f(zero, x) = x
f(succ(m),x) = f(m,h(x))
```

with an accumulator, we have that

```
f(m,x) = f'(m,x),
```

were function f′ is defined as:

```
f'(zero,x) = x
f'(succ(m),x) = h(f'(m,x)),
```

without an accumulator. Morris (1971) has a similar more schematic result.

Cooper presents us with a more complex scheme: the function h in the accumulator argument position depends on the recursion variable. That is, we have f defined thus:

```
f(zero,x) = x
f(succ(m),x) = f(m,h(m,x)).
```

We then have that

```
f(m,x) = f'(m,x),
```

where

```
f'(zero,x) = x
f'(succ(m),x) = h(m,f'(m,x)),
```

provided that h(m[1],h(m[2],x)) = h(m[2],h(m[1],x)).

This result is quite different from the preceding one: the equivalence between f and f′ is preserved for a certain class of interpretations only. Darlington and Burstall (1976) use this idea with a number of schemes in their program improving system. They make the point that more general results are of little practical utility.

In a private communication, Burstall (1974) proposes this transformation approach to the accumulator problem. That is, he proposes to use an array of schemes allowing the elimination of accumulators in the useful cases. In addition to the above schemes for natural numbers, he gives one for lists. If f is defined as:

```
f(nil,x) = x
f(cons(n,l),x) = f(l,h(n,x)),
```

then

$$f(l,x) = f'(rev(l),x),$$

where f' is defined as

$$f'(nil,x) = x$$
$$f'(cons(n,l),x) = h(n,f'(l,x)).$$

Presumably, all these transformations can also be made with the help of the more flexible method which Burstall and Darlington (1976) developed later and which does not appeal to schemes directly.

This alternative approach of reducing definitions instead of generalizing accumulators to variables appears to have some scope. Nonetheless, it also appears to be no less involved and difficult to apply than our generalization technique.

There is a middle of the road approach which does not do any reduction of definitions before the proofs or generalizations of accumulators to variables in the course of the proofs. Instead, accumulators are replaced, in the course of the proofs, by function applications whose rators are new function constants. These are defined so as to represent the general computation of the accumulators. That is, if accumulator x on the left of a definition becomes h(x) on the right, these new functions compute the value of h(...(h(x))...), n times, in terms of n. These definitions can be found by using schemes (Moore 1974) or be synthesized (Aubin 1975). Not surprisingly, these functions turn out to be similar to those obtained by reduction as above, but this method cannot easily find the special conditions attached to the equivalence result and thus, can lead to non-provable subgoals.

Take for example the reduction for lists given above. This method will find the definition of f′ in a given proof, but will ignore the fact that f=f′ only if the first argument of f′ is reversed. Moore tries to minimize this problem by writing similar definitions for accumulators on both sides of = or =>, hoping that this would result in an implicit and provable generalization of rev(l) to a new variable.

One could have thought that such a middle of the road method would have brought together the qualities of both reductions and generalizations. But in my view, it turns out to be less elegant than either. If the use of schemes appears sufficient in Moore's uniquely typed language, in a language with a richer type structure like ours, it is likely that the number of useful schemes will grow fast. And the problem of taking special conditions into account still remains. On the other hand, synthesizing function definitions without the help of schemes is difficult. This can be systematized, but how it can be done uniformly and efficiently in the course of a proof is still very much an open question. The difficulties in the mechanization of such a strategy are very much like those encountered by Burstall and Darlington (1976); they propose a semi-automatic solution to them.

CHAPTER 5

INDUCTION SUBGOALS

This chapter is devoted to the second part of the induction tactic. In the previous chapters, we saw how some of the variables of a goal were selected to do induction upon; generalizations had to be done _en route_, whenever possible, in order to facilitate the use of the induction hypotheses. We now want, on the one hand, to find the induction subgoals, given the list of induction variables. In particular, we need to find heuristically justified instantiations for the induction hypotheses; we may also wish to discard some hypotheses judged useless. On the other hand, we want to make use of the induction hypotheses which have been retained in order to actually help to find the proof of the goal. These two points constitute the two main sections of this chapter.

5.1 INDUCTION SUBGOALS

5.1.1 Extension To Formal System

Our induction and definition by k-recursion rules are two facets of the same reality: our domain is a free algebra generated from the empty set. So, it is not surprising that clues like the occurrence of a variable in a recursion argument position were

important for the selection of induction variables. Similarly, definitions of the function constants occurring in the induction goal are going to play a key role in the instantiation of the induction rule itself, and in the instantiation of the induction hypotheses.

For the moment, I would like to extend the way functions can be defined while retaining the analogy between induction and recursion. It is often more natural to write a definition by recursion from more than one base level (we will see why the word level is an appropriate one). For example,

```
ord(l) : bool <=
cases l [nil <= true |
         cons(n,nil) <= true |
         cons(m,cons(n,l)) <= m=<n & ord(cons(n,l))]
```

But this violates the syntax given in the first chapter, since, for one thing, there should be only one alternative of patterns for variables of type list and this definition has a case expression with three patterns bearing upon l. Indeed, type list is defined thus: [nil: | cons:nat,list] -> list, where terms of type nat denote natural numbers. So, the admissible format of our definitions need to be reexamined.

Some notions are useful to start with. I recall from chapter 2 that a structure of type tau is a term whose subterms of type tau have a constructor constant as rator and whose terms of other types are variables. The level of such a structure was defined as the length of the longest chain of nested reflexion constructor constants for type tau in the structure. A *substructure of a structure s of type tau* is simply a subterm of type tau of s. The *depth of a substructure t of a structure s* is the length of the chain of nested constructor constants from the leading rator of s to that of t

exclusively. Finally, an *open structure of level 1* is obtained from a structure of level greater than or equal to 1 by replacing all of its subterms of depth 1 by distinct variables. Note that a single variable is an open structure of level 0. I will often stick the qualification *closed* to structures which are definitely not open.

For example, recall the type definition [atom:nat | consx:sexpr,sexpr] ->sexpr and declare [sx[1] sx[2] | sx[3]] :bool. Then:

1. atom(n) is a closed structure of level 0
2. sx[1] is an open structure of level 0
3. consx(consx(atom(n[1]),atom(n[2])),atom(n[3])) is a closed structure of level 2
4. consx(consx(sx[1],sx[2]),atom(n[3])) is an open structure of level 2
5. *but* consx(consx(sx[1],sx[2]),sx[3]) is not a structure, either closed or open.

With these notions in mind, the syntax for case expressions which has been accepted up till now can be reinterpreted as follows: if tau is the type of the case variable x, then: (1) the admissible patterns for the case clauses are exactly the closed structures of level 0 and the open structures of level 1 of type tau, and (2) in the expression part of the case clause, any proper open substructures of the case clause pattern can occur in the argument position of x for the constant function being defined; these happen to be reflexion variables. For example, the single list of level 0 is nil, and the single open list of level 1 is cons(n,l) (variables can be renamed); these two patterns are those which have been permitted so far in case expressions. Moreover, nil has no proper substructures

and does not allow recursion, while cons(n,l) has the proper open substructure l which can occur as recursion variable.

Now, if we allow the admissible patterns to be exactly the closed structures of levels 0, ..., l-1, together with the open structures of level l (l=<l) of the type of the case variable, we obtain a generalized form of case expressions. Such definitions by cases are interpreted as k-recursive definitions from more than one base level. In effect, a progression by levels is the only one consistent with the way in which the carriers of our domain of interpretation are generated; the substructure ordering is also in agreement with the ordering over our domains as given in chapter 1.

One can check that the above definition of ord by cases is now syntactically correct; it is interpreted as

ord(nil) = true
ord(cons(n,nil)) = true
ord(cons(m,cons(n,l))) = n=<m & ord(cons(n,l))

How can we use such definitions without putting the consistency of the system in jeopardy? In fact, we already possess the answer to this question. One should recall from chapter 1 that the following principle of induction holds for the ordered set [S*;=<<], where S* is a product of carriers in the domain of interpretation and =<< is the lexicographic ordering on it:

If P(t*), for all t* in [S*;=<<],
whenever P(s*), for all s* in [S*;=<<] with s*<<t*,
then P(r*), for all r* in [S*;=<<].

So far, we have been limiting ourselves to the case where s* was an immediate predecessor of t* in [S*;=<<]; from now on, this restriction will be partially lifted.

If we reformulate the definition by k-recursion by allowing the function being defined to take <u>any</u> preceding values of its recursion arguments in its explicit definition, then the existence and uniqueness of such functions can be proved using the full structural induction principle; the proof follows the same line as in chapter 1. Such functions are said to be defined by course-of-values k-recursion. The point is that k-recursion from several base levels is a special case of course-of-values k-recursion. In effect, such a definition can be written as a definition giving on the one hand, a value to the function for closed recursion arguments of level 0, and the other hand, a value to the function for closed recursion arguments of levels 1, ..., l-1, and for open recursion arguments of level l in terms of values of the l preceding levels. In conclusion, functions such as <u>ord</u> can be used safely.

How can the nice correspondence between recursion and induction be maintained when such definitions are admitted? We need a rule of induction from more than one base level. This can similarly be obtained from the general principle of structural induction by an argument by cases on the c(x*)'s as to whether they are of levels 0, ..., l-1 and closed, or of level l and open. This justifies the validity of the actual rule used by the induction tactic:

```
u[1] ... u[n]
-------------
      u
```

where each u[i] is an implication of the form:

```
u [s[i,1] / z] & ... & u [s[i,m] / z] => u [s[i] / z].
```

There is precisely one u[i] for each structure s[i] of level 0, ..., l-1 and open structure s[i] of level l. For every u[i], the s[i,j]'s are precisely the open proper substructures of s[i].

This rule can be extended to cater for induction on any number of variables.

For lists, the rule of induction from several base levels can be instantiated to

```
u [nil /z]
u [cons(n,nil) /z]
u [l /z] & u [cons(n,l) / z] => u [cons(m,cons(n,l)) / z]
-----------------------------------------------------------
                          u
```

This is an induction from the null list and the list of length 1, which matches well definitions like that of ord given at the beginning of this section.

In conclusion, our induction tactic can make use of a general rule of induction from l base levels, which rule is analogous to definition by k-recursion from l base levels (1=<l).

5.1.2 General Method

In his thesis (1973), Moore gives a way of doing induction on any number of variables for list theory and a limited form of induction from two bases, but without incorporating the possibility of instantiating variables free in the induction hypotheses. (The type list is defined thus in his case: [nil: | cons:list,list] -> list.) However, in a later version of his prover (Moore 1974), induction from any number of bases is, I believe, included in a disguised form, as well as the possibility of variable instantiation.

In order to generate the induction subgoals, Boyer and Moore use a method which maps the structure of what they call a bomb list into the required terms. The bomb list of a goal contains information about how definitions fail to apply to the goal. In Moore's later version, the corresponding mechanism is directly based on function definitions.

I would argue that such techniques whereby induction subgoals are more or less directly constructed from function definitions do not constitute a sound approach. We have seen that in the case of the Boyer-Moore prover (as well as the present one), the induction rule is never applied as such: there is only an induction tactic which yields subgoals from a goal. What guarantee could Boyer and Moore offer that their induction tactic is indeed the inverse of their induction rule? Since their tactic is based on function definitions, one may wish to prove that it is consistent for any admissible definitions. This approach seems reasonable at first sight. However, in the case of Boyer and Moore the admissibility of definitions is not computable. In effect, they allow the theory to be extended by any total recursive functions. But totality is not computable, so admissibility is not either. In conclusion, the consistency of their tactic is not provable. Furthermore, the possibility of adding any total functions makes it very difficult to pinpoint which induction principle _exactly_ is used. Note that Brotz does not have this problem, since his language is not extensible.

In the present case, function definitions are constrained to be k-recursive, which ones have been proved to be total. So, one might be able to carry out the program proposed in the previous paragraph. However, I do not favour this approach. For two reasons.

One is pragmatic: despite the fact that k-recursiveness can be checked, I have not written a parser to this effect. This means that in practice, I am in the same uncertain situation concerning the totality of my definitions as Boyer and Moore are for theirs. Even though case expressions provide one with a disciplined way of writing k-recursive functions, they are obviously not fool proof.

A more important reason is one of method. My induction tactic is based on type definitions as one feels it should naturally be. All induction subgoals are generated, and for each of them, all induction hypotheses are considered for heuristic relevance, and are discarded or retained and instantiated accordingly. As we have seen in chapter 2, rejecting an induction hypothesis is consistent (by the weakening rule) and preserves the provability of the induction subgoal, if the goal is provable. It appears easier to convince oneself of the correctness of such a top-down method than of a bottom-up approach whereby induction subgoals are generated from information about function definitions.

Checking the admissibility of the definition of a type constant tau is straightforward. Rejection is immediate if tau has already been introduced. Otherwise, for each clause of the definition, we make sure that (1) the constructor constant has not been previously introduced, and (2) all type constants appearing in its argument positions have either been defined before or are tau itself; however, tau is not allowed to occur in all clauses. Note that the definition with zero clause is admissible.

This does not mean that the definitions of the function constants appearing in the induction goal have no part to play in the induction tactic; quite on the contrary, they are crucial. However, their participation will not go beyond giving information about (1) the number of base levels from which induction has to be done, and (2) the rejection, or the acceptance and instantiation of tentative induction hypotheses. Precisely which definitions can serve a useful purpose is given by the induction applications. We have seen that these are primary applications in the induction goal whose recursion arguments are either induction variables, or have a constructor constant as rator, or are induction applications themselves. Such function applications are important, because if the induction variables were replaced by structures, then the rule of definition by k-recursion would have some chance of being applicable to them. Since definitions are used only indirectly, this induction tactic remains consistent even if a user inadvertently introduces a function which is not k-recursive.

It is desirable at this point to show how the substitutions for the hypotheses and conclusions of the induction subgoals of a simple example are generated. We have ack(n,m)>0 as induction goal and there are two induction variables: n and m. The function constant ack is defined as:

```
ack(n,m) : nat <=
cases n [zero <= succ(m) |
         succ(n) <=
         cases m [zero <= ack(n,succ(zero)) |
                  succ(m) <= ack(n,ack(succ(n),m))]].
```

We start by generating the conclusion substitutions. We can find just one basic induction application in the goal, namely ack(n,m), and the definition of ack tells us that both of its recursion arguments are defined from one base level. So, we get as possible substitutions for n, the closed structure zero and the open structure, say, succ(n[1]); similarly, zero and succ(m[1]) can be substituted for m. This means that there are four conclusion substitutions: (1) [zero/n] [zero/m], (2) [zero/n] [succ(m[1])/m], (3) [succ(n[1])/n] [zero/m], and (4) [succ(n[1])/n] [succ(m[1])/m].

We next have to find zero or more hypothesis substitutions for each conclusion substitution, according to our lexicographic ordering:

1. We cannot derive any hypothesis substitution from the substitution [zero/n] [zero/m], since zero has no proper substructure
2. The substitution [zero/n] [succ(m[1])/m] yields one hypothesis substitution: [zero/n] [m[1]/m], but it is discarded since the definition of ack is not recursive for this case
3. From the third substitution, the first algorithm finds the hypothesis substitution [n[1]/n] and any term can be substituted for m; the definition of ack tells us to retain this substitution and to substitute succ(zero) for m
4. Finally, there are two substitutions generated from the conclusion substitution [succ(n[1])/n] [succ(n[2])/m]: (1) [n[1]/n] and any term for m, and (2) [succ(n[1])/n] [m[1]/m]; according to the definition of ack, both of them should be kept and m should be replaced by

ack(succ(n[1]),m[1]) in the first substitution.

The following sections explain in detail the algorithms by which such substitutions for the conclusions and hypotheses are generated. By applying these substitutions to the induction goal, and by bundling up the resulting terms with => and &, we easily obtain the induction subgoals themselves.

5.1.3 Generation Of Induction Conclusions

The induction tactic first finds the substitutions which have to be applied to the goal in order to constitute the induction conclusions.

We are given a list of induction variables taken from a goal. The first thing to discover for each of these variables is the number of base levels from which induction has to be done. This clue is given by the subclass of induction applications which I have called basic. These are induction applications whose arguments are all induction variables; their importance lies in the fact that they can certainly be simplified when the conclusion substitutions sought for are applied to the induction goal. When a function constant is introduced, we examine from how many base levels it is defined for each recursion argument. A list of numbers is thus remembered which gives the number of base levels from which the function constant is defined for each recursion argument respectively.

<u>Number of base levels corresponding to each induction variable</u>

We start by associating 1 to each induction variable: there is always at least one base level in any induction. For each basic induction application, each of its recursion arguments is an induction variable v by definition. On the property list of the function constant of its rator, there is also a number associated with this recursion argument position, giving how many base levels are required. If this number is greater than the number already associated with v, we replace the latter by the former. We terminate with a number of base levels greater than or equal to 1 associated to each induction variable. ||

Note how indirectly the information about definitions is fed to the induction tactic.

The second step consists in finding, for each induction variable v, all the structures to be substituted for v in one or the other of the conclusion substitutions, taking into account the number l of base levels associated with v. This is immediately given by our generalized induction rule. Let tau be the type of v. We need to replace v by all closed structures of level 0, ..., l-1, and by all open structures of type tau of level l of type tau. In chapter 2, we had an algorithm which gave the number of structures of a given type for any given level. We also had a main algorithm whereby any structure of a given index could be generated. So, these algorithms can be used to construct the required closed structures of levels 0, ..., l-1. But, what about open structures?

The algorithms of chapter 2 can be modified without pain to allow the generation of either closed or open structures.

<u>Number of closed or open structures of type tau of level l</u>

If l is equal to 0, the number of <u>closed</u> structures of level l is the number of constructor constants of reflexivity 0 in the definition of tau, while there is only one <u>open</u> structure of level 0. Otherwise, we set a variable n to 0. For each reflexion constructor constant c in the definition of tau, for each development l[1], ..., l[k] of l-1, where k is the reflexivity of c, we recursively compute the number of <u>closed</u> structures of level l[i], for all i (1=<i=<k and l[i]/=l-1), and the number of <u>closed</u> or <u>open</u> structures of level l[j], for all j (1=<j=<k and l[j]=l-1), according to whether we are currently seeking for a closed or open structure (there is at least one such j by definition of development); we make the product of all those numbers, and add it to n. The final value of n is the result expected for. ||

The two modifications one has to bring to the main algorithm in order to generate a closed or open structure of any index are analogous to those supplied in the above algorithm and will not be given in detail here.

These algorithms provide us with the necessary tool to successfully carry out the second step of the present procedure.

<u>Possible substitutions for the induction variables</u>

For each induction variable v, if l is the number of base levels associated with v, we generate all closed structures of level 0 ,..., l-1 and all open structures of level l, and associate them with v. ||

Thirdly, and finally, we construct the actual substitutions for the conclusions.

Conclusion substitutions

We constitute all substitutions such that the variables of the components are exactly the induction variables and the term of each component is a structure associated with the variable, as found in the previous part. ||

Example 1

The induction subgoal is a=>a and the induction variable, a. The function constant => is defined by recursion on its first argument; so the goal itself is the only basic induction application. Clearly, => is defined from only one base level. So, we have to generate all closed structures of level 0: these are true and false. But there are no structures of level greater than 0; hence, there are only two possible conclusion substitutions: [true/a] and [false/a].

This simple theorem (and somewhat fictitious, since it is part of the basic system amplification) is given to illustrate that what is usually called case analysis is an instance of the induction rule in the present system. ||

Example 2

We want to show that ord(l)=>ord(tolist(n,l)) by induction on l. This time, there are two basic induction applications, namely, ord(l) and tolist(n,l). The function constant tolist is defined from one base level on its second argument, but ord is defined from two base levels. So, the latter outweighs the former; we need the

Thirdly, and finally, we construct the actual substitutions for the conclusions.

Conclusion substitutions

We constitute all substitutions such that the variables of the components are exactly the induction variables and the term of each component is a structure associated with the variable, as found in the previous part. ||

Example 1

The induction subgoal is a=>a and the induction variable, a. The function constant => is defined by recursion on its first argument; so the goal itself is the only basic induction application. Clearly, => is defined from only one base level. So, we have to generate all closed structures of level 0: these are true and false. But there are no structures of level greater than 0; hence, there are only two possible conclusion substitutions: [true/a] and [false/a].

This simple theorem (and somewhat fictitious, since it is part of the basic system amplification) is given to illustrate that what is usually called case analysis is an instance of the induction rule in the present system. ||

Example 2

We want to show that ord(l)=>ord(tolist(n,l)) by induction on l. This time, there are two basic induction applications, namely, ord(l) and tolist(n,l). The function constant tolist is defined from one base level on its second argument, but ord is defined from two base levels. So, the latter outweighs the former; we need the

closed lists nil and cons(n[1],nil) as well as the open list cons(n[3],cons(n[2],l[1])). These three structures immediately yield the conclusion substitutions for this problem. ||

Example 3

We conjecture that mem(n,l)=>mem(n,union2(k,l)) and we want to do the proof by induction on k. We find a single basic induction application for which one base level is needed. So, the conclusion substitutions are [nil/k] and [cons(n[1],k[1])/k]. ||

Example 4

The goal states the correctness of a compiling algorithm for expressions (a variant of the McCarthy-Painter lemma in Milner and Weyrauch 1972; see also Burstall 1969):

```
mt(comp(e),str)
=
mkstore(stof(str),
        push(mse(e,stof(str)),pdof(str)))
```

In other words, we get the same store if we compile an expression and interpret the resulting program, given a store, as if we interpret the expression with the state of the given store and push the result down on the stack of the store, leaving its state unchanged. Induction is on e and the two basic induction applications comp(e) and mse(e,stof(str)) demand one base level. So, we get two conclusion substitutions: [simple(na[1])/e] and [compound(o[1],e[1],e[2])/e]. ||

5.1.4 Generation Of Induction Hypotheses

For each conclusion substitution, we have to find zero or more hypothesis substitutions. As already mentioned, we will consider all possible uninstantiated such substitutions: some will be discarded, some will be retained and instantiated. Induction applications will play a crucial, but again indirect, role in the process.

A word of warning: in the sequel, I will often talk of instantiating a variable in an induction hypothesis; strictly speaking, what is instantiated is the metavariable over terms by which the variable has been replaced in the hypothesis. But this will not create any confusion.

Consider any conclusion substitution: what does our induction rule say about substitutions for the hypotheses? Suppose [s/x] [t/y] are the two first components of the conclusion substitution. Variables x and y are the two first induction variables and certainly occur in the induction goal. In order to construct a hypothesis of this conclusion, we can replace x in the goal by any proper substructures of s, and replace y and the other variables of the goal by any term; or we can replace x by s itself, but replace y by any proper substructures of t and all other induction variables by any term; and so on, with the other components of the conclusion substitution. Thus we obtain all hypothesis substitutions for a conclusion substitution. The induction applications will come into the picture to serve two purposes: (1) to reject a hypothesis substitution if no use can be foreseen for it, and (2) to find relevant instances for the variables

which can be replaced by any term. They are not used to find the structures by which the other variables have to be replaced. So consistency is maintained.

The algorithm has then two intermixed parts: the generation of uninstantiated hypothesis substitutions, and their selection and instantiation, if necessary.

Generation of uninstantiated hypothesis substitutions

For each conclusion substitution, we start with the empty hypothesis substitution. For each component of the conclusion substitution, we find all substructures of the term of the component. From each of these structures and the variable of the component, we build a substitution component: it is added to the current hypothesis substitution to form a tentative substitution which can be rejected, or accepted and instantiated, possibly in several ways (see second part of algorithm). After all substructures have been considered, we definitely add to the current hypothesis substitution the current component of the conclusion substitution before passing on to the next component of this substitution. We terminate with a list of hypothesis substitution lists, one for each conclusion substitution. ||

The second part of the algorithm, which is actually done concurrently with the first one, accepts or rejects an hypothesis substitution. The principle underlying this selection is the same as the one which has been consistently followed throughout the previous chapters: we want the best possible match between the hypotheses and the conclusions of the induction subgoals. In effect, the choice of the induction variables has been made so that definitions which were

not applicable to the induction goal becomes applicable to the conclusions of the induction subgoals. It is hoped that the induction conclusions, once simplified, will match the induction hypotheses more or less perfectly well. This is how the analogy between recursion and induction is useful to us.

The induction applications have an obvious role to play in this matching. In fact, two not necessarily disjoint subclasses of them will be useful: the basic induction applications encountered in the previous section, and the induction applications with an accumulator which is a variable. With the help of the first ones, we select hypothesis substitutions and find relevant instances for the induction variables which can be replaced by any term; with the second ones, we instantiate variable accumulators, if possible.

In the induction conclusion, the induction variables will be replaced by some structures and the application corresponding to a basic induction application of the goal will necessarily be simplified. Because definitions are recursive, some subterms of the simplified application will have the same rator as the application before simplification. We wish those subterms to match the application corresponding to the basic induction application, this time, in the induction hypotheses: this is how relevant instances are found for the free induction variables. Next, an application with an accumulator which is a variable may be simplified. Again, some subterms of the result may possibly share their rators with the non-simplified application. Now, we want the accumulators of those subterms to match the corresponding accumulators in the hypotheses (provided the recursion arguments match), which gives instances for the variable accumulators.

Suppose, for instance, that rev2(app(j,k),l) is a subterm of an induction goal on j. If we replace j by cons(n[l],j[l]), the inner induction application (a basic one) simplifies to cons(n[l]),app(j[l]),k)); by matching, we know that the hypothesis substitution [j[l]/j] should be retained. As it turns out, the outer induction application can also be simplified; we obtain rev2(app(j[l],k),cons(n[l],l)). So the variable accumulator l should be replaced by cons(n[l],l) in the induction hypothesis.

The decision of not considering the terms occurring in non-recursion argument positions for instantiation is natural. In effect, those terms correspond to parameters which remain fixed in the definitions, and consequently nothing should be gained by instantiating them, especially if a separation of variables has been achieved as explained in chapter 3. Finally, the limitation put on accumulators to be variables is consistent with the generalization tactic which tries to replace all non-variable accumulators by variables.

This is the rationale behind the following algorithm which starts with an uninstantiated hypothesis substitution.

Selection and instantiation of hypothesis substitutions

We apply the conclusion substitution to each induction application in the two subclasses considered above. We simplify the result but without normalizing it (a distinction will be drawn in the next chapter between pure simplification and normalization), and we collect all applications of the simplified term which share their rators with the induction application: they can all potentially give birth to an instantiated hypothesis substitution.

For each of these applications, we successively try to match the recursion terms of the induction application with the recursion terms of the application, taking into account the fact that some of the variables may already be bound by the uninstantiated hypothesis substitution under consideration. If the match fails, we reject the hypothesis substitution as being irrelevant for this application. Otherwise, and if the recursion terms were variables, we append the substitution found by the match to the hypothesis substitution: this constitutes a good instantiated substitution which we retain. Finally, if the induction application has an accumulator and if it is a variable which matches the accumulator of the application under consideration, we append the resulting substitution to the instantiated hypothesis substitution found above.

We terminate with a list of instantiated hypothesis substitutions. ||

As one might have guessed, the hypothesis substitutions gathered for each conclusion will be quite redundant in general. This redundancy can be removed by noticing the following: if two substitutions s[1] and s[2] have their n first components equal, and if the rest of their components are mutually disjoint, they can be replaced by a substitution whose n first components are the n first components of any one of s[1] and s[2] and whose next components are obtained by concatenation of, say, s[1] and s[2], less their n first components. Two substitutions are disjoint if the variable of any component of one does not occur as the variable or in the term of any component of the other. Moreover, any component of the form [x/x] can be removed from a substitution.

I do not pretend that this method of instantiating hypotheses provides one with exactly what is needed for the proof of a subgoal. If anything, however, it errs on the side of safety. By considering all induction applications, it will sometime find hypotheses which are not strictly necessary to the proof. But this is obviously much better than missing out a crucial one.

I will now show how hypothesis substitutions are found for the examples of the previous section.

Example 1

We have found two conclusion substitutions for the goal a=>a: [true/a] and [false/a]. None of them have corresponding hypothesis substitutions since neither true nor false contain any proper substructures. ||

Example 2

The third subgoal of ord(l)=>ord(tolist(l)) is the most interesting. Two hypothesis substitutions are possible with its conclusion substitution [cons(n[3],cons(n[2],l[l]))/l]; they are [cons(n[2],l[l])/l] and [l[l]/l]. The first one is retained because of the definition of ord; the second, because of the definition of tolist. ||

Example 3

With the goal mem(n,l)=>mem(n,union2(k,l)), an accumulator has to be instantiated in the induction application union2(k,l), namely l. This particular example illustrates a situation where the accumulator has in fact to take more than one instance.

We are chiefly concerned with the non-trivial conclusion substitution [cons(n[1],k[1]) / k]. The first part of the algorithm gives the tentative hypothesis substitution [k[1] / k] with any term for l.

The second part applies the conclusion substitution to the induction application union2(k,l) and simplifies the result, yielding:

cond(mem(n[1],l), union2(k[1],l), union2(k[1],cons(n[1],l))).

The second and third arguments of the conditional share their rator with the original induction application. By matching, we discover not only that the tentative hypothesis substitution should be retained, but also that l should have two different instances: l itself and cons(n[1],l). We finally get:

[k[1] / k] [l / l]

and

[k[1] / k] [cons(n[1],l) /l].

At the end, the trivial component [l/l] is eliminated. ||

<u>Example 4</u>

To prove this theorem, we must have the following equality as simplification rule:

mt(approg(pr[1],pr[2]),str)
=
mt(pr[2],mt(pr[1],str)).

It is an instance of a general property of functions with accumulators. We start with the conclusion substitutions [simple(na[1])/e] and [compound(o[1],e[1],e[2])/e]. Only the second one is of interest. The possible hypothesis substitutions are [e[1]/e] and [e[2]/e] with any term substituted for the variable

accumulator str in each case. We have three relevant induction applications: comp(e), mse(e,stof(str)), and mt(comp(e),str). The definitions of comp and mse require that both induction substitutions should be retained. The third induction application will give us instances for str. If we appply the conclusion substitution to mt(comp(e),str) and simplify the result, we get:

```
mkstore(
 stof(mt(comp(e[2]),mt(comp(e[1]),str))),
 push(
  apply(
     mo(o[1]),
     top(bottom(pdof(mt(comp(e[2]),mt(comp(e[1]),str)))),
     top(pdof(mt(comp(e[2]),mt(comp(e[1]),str))))),
  bottom(bottom(pdof(mt(comp(e[2]),mt(comp(e[1]),str))))))
```

If we remove the noise from this term, we are left with mt(comp(e[2]),mt(comp(e[1]),str)). By matching, we find that the only instance of str compatible with e[1] is str itself, while for e[2], str should be instantiated to mt(comp(e[1]),str). In conclusion, we get two hypothesis substitutions: [e[1]/e] and [e[2]/e] [mt(comp(e[1]),str)/str].

All subgoals of examples 1, 2, 3, and 4 are easily provable. The full definition and correctness proof of the compiling algorithm is given in section 3 of appendix 3.

5.2 USE OF INDUCTION HYPOTHESES

Selection of induction variables, generalization, selection and instantiation of induction variables, everything that has been discussed so far tended toward the best possible match between the induction hypotheses and conclusions. How is this going to be made to profit? The method used by this prover is an adaptation of what

has already been worked out by Brotz and especially by Boyer and Moore.

One must recall at this point the normal form of a term: an implication whose antecedent is a conjunction and whose consequent, a disjunction. The following techniques are applied to any such implication and not only to induction subgoals; this creates problems which will be discussed as we go along.

5.2.1 Replacement

The first way of using a hypothesis is not really replacement. If a term occurs as member of both the antecedent and consequent of an implication, this one is immediately solved. This expeditive method is applied by the simplification tactic. The rule used is the generalized reflexivity of =>, i.e. (a[1] & ... & a[n] => c[1] v ... v c[m]) = true, if a[i]=c[j] for some i and j (1=<i=<n and 1=<j=<m). Exactly how such simplification rules are used will be discussed in the next chapter. This first situation is the best which can arise in the case of an induction subgoal: the whole of a hypothesis matches the whole of the conclusion.

When this is not possible, the replacement tactic has to come into action. This tactic makes use of the equality substitutivity rule with an antecedent. For those members of the antecedent which are equalities, it tries to replace the right by the left-hand side (or vice versa according to occurrence and complexity) in one or more members of the consequent.

A simple measure of complexity is used, which resembles measures based on symbols. The complexity of a variable is 1; the complexity of a function application is equal to the sum of the complexity of each of its argument (multiplied by 10 for accumulators which are not variables), plus 1. This measure is biased against non-variable accumulators because, as we have seen, these require an expensive kind of generalization.

Replacement

For each member of the antecedent, for each member of the consequent, we first try a careful mode of replacement. We either try to replace the left by the right-hand side of the antecedent member into the *left*-hand side of the consequent member, or the right by the left-hand side of the antecedent member in the *right*-hand side of the consequent member, according to occurrence and complexity. That is, we try to do one of 1, 2, and 3:

1. We reduce $s=t \Rightarrow u[s/z]=w$ to $s=t \Rightarrow u[t/z]=w$, where z occurs in u, and s and t do not occur in u and w respectively
2. We reduce $s=t \Rightarrow u=w[t/z]$ to $s=t \Rightarrow u=w[s/z]$, where z occurs in w, and s and t do not occur in u and w respectively
3. According to whether $\text{complexity}(t) =< \text{complexity}(s)$ or not, we reduce $s=t \Rightarrow u[s/z]=w[t/z]$ to $s=t \Rightarrow u[t/z]=w[t/z]$ or $s=t \Rightarrow u[s/z]=w[s/z]$, where z occurs in u and w, and s and t do not occur in u and w respectively.

If none of 1, 2, and 3 is applicable, then we try to replace one side of the antecedent member in the consequent member taken as a whole, according to occurrence and complexity. Again, there are three alternatives:

1. We reduce s=t => u[s/z] to s=t => u[t/z], where z occurs in u, and s and t do not

2. We reduce s=t => u[t/z] to s=t => u[s/z], under the same provisos

3. According to whether complexity(t)=<complexity(s) or not, we reduce s=t => u[s/y][t/z] to s=t => u[t/y][t/z] or s=t => u[s/y][s/z], where z and y occur in u, z does not occur in s or t, and s and t do not occur in u.

If none of these three cases is applicable, the whole of the replacement tactic is not applicable. ||

In his thesis (1973), Moore gives an excellent account of why this way of making equality substitution is at all helpful when the implication under consideration is a simplified induction subgoal coming from a goal which is an equality. I will only adapt it to the present context. The reason why the replacement tactic has to be used in the first place is that only part of an induction hypothesis matches part of the conclusion; otherwise, in effect, the whole induction subgoal could be reduced to true by simplification. In particular, we are interested in the case where a subterm which matches part of the conclusion occurs as the whole side of an equality. Take, for instance, the simplified induction subgoal s=t => u[s/z], where u is an equality. We can positively say that the hypothesis and the conclusion agree about s; but we also have the negative bit of information that they fail to match about t. When we replace s by t in the consequent, we get a term u[t/z] which is of the genre of t, as Moore puts it. And since t was precisely the term for which the match was missed, we now have the chance of smoothing away this difficulty.

Example

One of the simplified induction goals of the commutativity of * (times) is:

```
n*m = m*n
=> n*succ(m) = n+(m*n).
```

Induction was on m, but n would have been an equally good choice. So, in a sense, it is not surprising that the hypothesis and the conclusion only match for m, that is, only the right of the hypothesis recurs in the conclusion. Replacement reduces this implication to

```
n*m = m*n
=> n*(succ(m)) = n+(n*m). ||
```

This line of reasoning applies all the way to Brotz's equation calculus. However, with a logic like that of Boyer and Moore and the present one, it has to be revised and modified to cater for the situation where the induction is not an equality but e.g. an implication itself. In this case, after normalization of the induction subgoal, parts of the hypotheses and the conclusion can appear in both the antecedents and consequents of the resulting subgoals. For instance, an induction subgoal of the form (s=>t) => (u=>w) is normalized to t & u => w and u => s v w. If s=>t and u=>v come themselves from an earlier induction subgoal as often the case, then t may be the same as v, and u, the same as s; or else, t may be used for replacement in v, and u, in s. However, this is rather unlikely to be possible when the implication is an arbitrary one.

Moore's argument about equality can be extended to a similar argument about any relation, using the generalized notion of substitutivity introduced in section 3 of chapter 2. I will illustrate this with the relation =>. We have seen that the rules

$$\frac{(s\Rightarrow t) \Rightarrow (u\Rightarrow s)}{(s\Rightarrow t) \Rightarrow (u\Rightarrow t)} \qquad \frac{(s\Rightarrow t) \Rightarrow (t\Rightarrow u)}{(s\Rightarrow t) \Rightarrow (s\Rightarrow u)}$$

are derivable from the transitivity of =>. They can be generalized so that the antecedents and consequents of the implications involved in the substitutions are conjunctions and disjunctions respectively. If we first normalize the hypotheses and the conclusion of an induction subgoal separately, we can avoid the disruption of the hypothesis-conclusion pattern, in particular when the induction goal is an implication. Thus, we are in a position to try to apply a replacement tactic making use of the implication substitutivity rules. The freedom of substitution is a bit hampered in comparison with equaliy: not just any term can be replaced but only antecedent or consequent members. Moreover, substitutivity of equality is reversible because of the symmetry of =; this property is not shared by =>, so that the result of the tactic has to be checked for provability. One can think of replacement tactics for other relations as well. This generalized replacement for relations other than equality is but a proposal and has not been implemented and experimented with in the present prover.

However, there is a case for which Moore's argument does not carry up. The replacement tactic is well justified, and works well, when used within induction subgoals. But what about replacement with any odd equality? It appears that using the substitutivity of equalities not constrained within induction demands a different and new approach which has not been explored in depth by either Boyer and Moore or by myself. Brown (1976a), however, uses a replacement tactic which deals with such equalities; it is based on the rule

```
t=x => u[t/z]
-------------
t=x => u[x/z]
```

where variable x does not occur in term t. The symmetry of equality gives a twin rule.

5.2.2 Strengthening

A strengthening tactic is actually used concurrently with replacement. In effect, the antecedent members of an implication which are involved in replacement are discarded from the antecedent. In other words, the implication is strengthened, using the inverse of the weakening rule.

In the case of an induction subgoal, we have seen that it is always safe to remove a hypothesis; that is, if the induction goal is provable, any induction hypothesis can be rejected without altering the provability of the induction subgoal. So, no checking is necessary. But what tells us that the new amputated subgoal will be any easier to prove?

Again Moore gives a nice answer to this question. The replacement tactic has reduced the induction subgoal s=t => u[s/z] to s=>t => u[t/z], say. This was reasonable on the ground that the hypothesis and the conclusion match for s but not for t. The resulting consequent u[t/z] is now of the genre of t, which constitutes the next difficulty to tackle. Nothing can be gained by keeping s=t around. Both the positive information about s and the negative information about t have been used in the replacement, and the equality can only be a nuisance to other tactics. Indeed, we have seen earlier that antecedent and consequent members should be

discarded whenever possible in order to keep down the complexity of the goals.

I should add to my credit that the concept of strengthening is original to this prover and does not appear explicitly anywhere in the work of Boyer and Moore. In their prover, some induction hypotheses may be implicitly discarded in the formation of induction subgoals. However, after having used an equality for replacement in an implication, they do not reject it, but retain it in the new subgoal in a "hidden" form, inaccessible to other tactics; their purpose is to formally preserve the equivalence of the subgoals before and after replacement. This is a strange design decision since on the other hand, they do not hesitate in using the generalization tactic which cannot preserve equivalence.

For my part, I prefer to explicitly throw away an equality involved in replacement: if the rejected equality is an induction hypothesis, then strengthening does preserve equivalence anyway; otherwise, checking can be used to make sure that the new subgoal is provable (but this has not been incorporated in the present prover yet). This approach seems better than maintaining an artificial formal equivalence. (Actually, Boyer and Moore also use a replacement strategy within their simplification tactic. The equalities are then neither discarded nor hidden, but explicitly retained; this is quite justifiable in this case, since preserving equivalence is a desirable property of simplification.)

Example

The following subgoal of the commutativity of * was obtained after replacement:

```
n*m = m*n
=> n*succ(m) = n+(n*m).
```

We strengthen it to

```
n*succ(m) = n+(n*m).
```

Now, n becomes the natural choice of induction variable. ||

However, this application of the strengthening tactic suffers from the same problem as its companion replacement tactic. The argument is well justified when the implication is a simplified induction subgoal. Otherwise, the rejected equality not being constrained within induction, the strengthened goal may not be provable. Thus, checking becomes necessary. Brown (1976a), who also adds a strengthening part to his replacement tactic, claims that under the conditions stated in the previous section, weakening becomes reversible. In other words, his replacement-strengthening tactic can be used without checking, which is rather nice.

In conclusion, the Boyer-Moore type of replacement-strengthening tactic is well justified with induction goals which are equalities. With other relations, and in particular implication, a generalized replacement tactic could be used with profit. Finally, replacement with equalities (and other relations for that matter) which are not constrained within induction requires a new approach. As implemented, the present replacement-strengthening tactic suffers from the same deficiencies as that of Boyer and Moore.

CHAPTER 6

SIMPLIFICATION AND OTHER STRATEGIES

The tactics which have been studied so far are all more or less directly related to induction. There are a few others not so much dependent on it. The most important of them is simplification, but I will first talk briefly about splitting and contraction.

6.1 SPLITTING

In section 2.4, we discussed the question of representation of our search space. At a certain level, we can view this search space as an and-or-tree of terms. But and-or-trees are themselves in the middle of a spectrum of equivalent representations ranging from and-trees of disjunctive nodes to or-trees of conjunctive nodes. We said that this prover explicitly split disjunctive nodes (but behind the scene) except for the connective v. This exception is reflected in the fact that the implication consequents of a term in normal form can have more than one disjunct as members.

From the point of view of splitting conjunctions, we can divide strategies into two classes: linear and non-linear ones. The former do not split conjunctions while the latter do. This prover falls in the second category. The splitting tactic is based on the conjunction rule and does not require checking. Among many others,

Bledsoe (1971) has advocated non-linear strategies. It helps keeping goals simple, which is one of our objectives; the overall efficiency of the theorem-prover is that much increased. From the user's point of view, the proofs are easier to read and possible lines of attack easier to assess in the case of a man-machine system. Brotz (1974) also splits conjunctions.

On the other side, Boyer and Moore (1975) had a linear strategy. A goal was a conjunction and tactics were applied to it until it reduced to true. However, the induction tactic would only select the first conjunct as induction subgoal, but without explicitly separating it from the remaining conjuncts.

The challenge facing both linear and non-linear strategies is to avoid duplication of work. That is, a subgoal should be solved once for all and be recognized as such should it recur in another part of the search space. (I will leave aside the question of the same subgoal occurring more than once on different or-branches since this is not applicable to the present prover.)

One must admit that linear strategies start with an advantage over non-linear ones. Since they do not split conjunctions, if a subgoal appears twice at the same time in the conjunctive goal, both occurrences can easily be identified by simplification. For example, the second and third induction subgoal of the theorem

```
ord(1)
=> ord(tolist(n,1))
```

both reduce to n=<m v m=<n. With a linear strategy like Boyer and Moore's, both subgoals are immediately identified. But with the present strategy which splits induction subgoals, the term in question appears on separate branches and have to be solved twice.

However, linear strategies give only part of the answer to a problem which requires a more general solution. In effect, even a linear strategy cannot identify the recurrence of subgoals at two _different_ moments in the conjunctive goal. So, in the end, both types of strategies are on the same foot: they both need some mechanism for remembering the subgoals which have been solved so far for any point of the search. The typical problem in this case is the number of subgoals which may become overwhelmingly large; one needs a selective memory.

The present prover does not have such a recollection mechanism to support its non-linear strategy. This turns out to be sometimes a grave defect. However, it is possible to give some indication as to what such a strategy would be like. We have seen how useful the context of induction is in order to constrain many strategies otherwise explosive: generalization, strengthening, instantiation of free variables, replacement. I submit that in such an induction-based theorem-prover, the only goals worth remembering are the induction goals, that is, those terms which no other tactics are applicable to, except induction. This cuts down considerably on the number of solved goals one would normally have to memorize in the absence of any other criteria. But since induction goals are generally implications (our normal form after splitting), we need a simplification tactic which can make use of such terms. This is not possible with the present strategy which can deal with equalities only. More will be said about this in a subsequent section.

6.2 CONTRACTION

Contraction is a simple tactic based on the third version of equality substitutivity derived in chapter 2, namely,

```
      t=s
-------------
w[t/x]=w[s/x].
```

In actual fact, the context of application of this tactic is more constrained than what this rule allows for. We limit the variable x to occur as an argument of the outermost function application in w. In other words, we reduce a goal of the form:

```
f(u[1], ..., s, ..., u[n])
=
f(u[1], ..., t, ..., u[n]),
```

where f is a function constant, to the subgoal

```
s=t.
```

On the other hand, as suggested in chapter 2, a relaxed form of the above rule can be derived:

```
u[1] & ... & u[n] => w[1] v ... v      s=t       v ... v w[m]
-------------------------------------------------------------
u[1] & ... & u[n] => w[1] v ... v w[s/x]=w[t/x] v ... v w[m]
```

So, finally, the contraction tactic applies to any consequent member of a term in normal form.

The subgoals obtained by contraction are not necessarily provable even though they stand a good chance of being so. For example, n*zero=m*zero cannot be contracted to n=m. Consequently, they must be checked for non-provability.

Here follows the algorithm:

Contraction

The algorithm has two parts; we start by assigning the goal to a program variable t.

Part 1

If t is a variable, we fail. Otherwise, if t is an equality, we apply part 2 to t. Otherwise, if t is an implication, we recursively apply part 1 to each of its consequent members (note: a disjunction is an implication with an empty antecedent). Otherwise, we fail.

Part 2

If one side of equality t is a variable, we fail. Otherwise, unless both sides have identical rators and identical arguments two by two, except for exactly one pair, we also fail. Otherwise, we make a new equality s with this dissimilar pair of terms. We replace t by s in the goal. Unless the resulting subgoal is checked non-provable, it becomes itself the new goal. We then recursively apply part 2 to equality s.

We terminate either with a failure to apply the tactic or with a new subgoal. ||

In this prover, contraction was originally designed as a useful tool in proving properties of functions with accumulators. The idea was that if two function applications were identical except for their accumulators, then these should be equal since they hold the result of the computation. This is exemplified in the proof of the associativity of times2. This function is defined thus:

```
times2(m,n) : nat <= times2a(m,n,zero)

times2a(m,n,p) : nat <=
cases m [zero <= p |
        succ(m) <= times2a(m,n,n+p)]
```

After one generalization and induction, the associativity property is reduced to

```
    times2a(m[1],
            times2a(n,p,zero),
            times2a(n,p,zero) + times2a(n[1],p,zero))
    =
    times2a(m[1],
            times2a(n,p,zero),
            times2a(n+n[1],p,zero)).
```

This equality is then contracted to

```
    times2a(n,p,zero) + times2a(n[1],p,zero)
    =
    times2a(n+n[1],p,zero)
```

which is provable. See appendix 3 for the full proof.

It soon became clear, however, that contraction could also be used with profit in any equality. Brotz (1974) has a similar tactic in his arithmetic prover which he says to be a form of generalization. Brown (1976a, 1976b) also makes use of a contraction tactic called "plus cancellation rule"; it is limited to applications whose rators are the function constant + and incorporates in it the associativity and commutativity properties of this operator. The output of this constrained contraction is always interdeducible with the input and needs no checking. This is particular to the function +.

In conclusion, contraction generally helps to keep goals simple, and in fact, plays a quite crucial role in proofs involving accumulators.

6.3 SIMPLIFICATION

Much attention has been paid to writing an efficient simplifier even if this tactic is not so closely related to induction. In effect, a good simplification tactic is of paramount importance to any symbol manipulation system. The simplification problem splits into three subproblems: (1) one of logical equivalence between terms before and after simplification, (2) one of complexity measure for terms, and (3) one of selection: what to replace by what in the terms to be simplified. The first section studies questions 1 and 2; question 3 is the subject of the whole of the second section.

6.3.1 Equivalence And Complexity

The first characteristic of a simplification tactic is that it should change the expressions of terms but not their meanings. This is, at least, the commonly accepted view. Simplification thus falls into the second of Polya's categories about working from a goal: we replace the goal by a logically equivalent subgoal.

This equivalence requirement is fulfilled in the present tactic. In effect, simplification replaces a term $w[s/x]$ by $w[t/x]$, provided that $s=t$. This is abstractly justified by the first form of equalitivity substitutivity derived in chapter 2. This rule was, in a sense, reversible since equality was symmetric. The term $s=t$ (in fact, a generalization of it, since $s=t$ will be an instance of a more general equality) is called a <u>simplification rule</u>. This phrase is appropriate since such equalities used for simplification are

theorems in a amplification of the basic formal system and in that sense, are derived rules without hypotheses.

But of course, there would be no point in using this tactic if the equivalent subgoal was not thought to be easier to solve. This is where the term simplification itself may be misleading. Simplicity must be measured by means of some yardstick. For terms, symbol complexity appears to be the first which comes to mind and the easiest to assess objectively.Term s is said to be simpler than term t in t=s if s has fewer syntactic constructs than t for all values of their variables. But from experience and according to our objectives, we know that other measures of simplicity can also be appropriate: we may want to have terms which are easy to comprehend by the user, or which can be efficiently manipulated by machine, etc. One may also have more eclectic tastes since in general, the simplest terms according to one measure or the other will not coincide precisely. (See Moses (1971) for a fuller discussion.)

I will first consider a subclass of simplification rules which may be called <u>pure simplification rules</u>. These are rules whose right-hand sides are definitely simpler than their left-hand sides from a pure symbol complexity point of view. We have seen in chapter 1 that the language the tactics deal with and the user sees is a amplification if the original language. The prospective user has in fact no direct control over this basic amplification. Part of it was constituted of the inference rules derived in chapter 2; we now add to it the following pure simplification rules:

1. (a = true) = true
2. (x = x) = true
3. cond(a, x, x) = x

4. cond(a, true, false) = a
5. cond(a, s, t) = cond(a, s[true/a], t[false/a])
6. not(not(a)) = a
7. a v true = true
8. a v false = a
9. a v a = a
10. a & true = a
11. a & false = false
12. a & a = a
13. (a => true) = true
14. (a => a) = true

The user can enlarge this basic list with more equalities of his choice.

Symbol complexity itself is, in fact, not often used in its purest form. Brotz (1974) imposes an additional lexical ordering on variables. He also has a more or less subjective ordering on function constants, e.g. + is less complex than -, etc. In this prover, our complexity measure is biased against non-variable accumulators.

At any rate, symbol complexity is not our sole criterion. Boolean terms are put into normal form, even if the normalized terms have more syntactic constructs that the terms they originate from. Such other measures of complexity are almost always present in mechanical theorem-proving on top of some symbol complexity measure. For example, Boyer and Moore (1975) used a normal form for conditional expressions.

I recall from chapter 1 that in this prover, a term in normal form satisfies the following scheme:

```
(a[1,1] & ... & a[1,k[1]]) => (c[1,1] v ... v c[1,m[1]])
& ... &
(a[n,1] & ... & a[n,k[n]]) => (c[n,1] v ... v c[n,m[n]])
```

(0=<n, 0=<k[i], and 0=<m[i]), where a[i,j] and c[i,j] are terms without connectives and conditionals.

There are good reasons for using a normal form, and this one in particular. Firstly, we want the prospective user to be able to read proofs relatively easily. Even if it is not a man-machine system, readable solutions are important, since we could guarantee only a certain level of confidence in the generated proofs. The sceptical user can then follow the proof more easily. Secondly, it is clear that no tactics can work reasonably smoothly and uniformly unless presented with terms in some sort of normal form. As already mentioned, the present normal form was inspired from a sequent calculus. It is easy to decipher (more than Boyer and Moore's conditionals); it lends itself naturally to efficient backward search.

So, the third part of our system's basic amplification consists of a number of normalization rules:

```
 1. (a & b) & c = a & (b & c)
 2. (a v b) v c = a v (b v c)
 3. f(x[1], ..., cond(a,x,y), ..., x[n])
    = cond(a, f(x[1], ..., x, ..., x[n]),
              f(x[1], ..., y, ..., x[n]),
    where f is some function constant
 4. a => (b => c) = a & b => c
 5. a => not(b) = a & b => false
 6. not(a) => b = a v b
 7. (a => b) => c = (b => c) & (a v c)
 8. a => b & c = (a => b) & (a => c)
 9. a v b => c = (a => c) & (b => c)
10. a => cond(b, c, d) = (a & b => c) & (a => b v d)
11. a => (b=c) = (a & b => c) & (a & c => b)
12. (a = b) => c = (a & b => c) & (a v b v c)
13. cond(a, b, c) => d
    = (a & b => d) & (b & c => d) & (c => a v d)
```

Clearly, the right-hand sides of these rules are sometimes more complex than their left-hand sides as regards the number of syntactic constructs. The user can add to these, but this has proved to be less useful than adding pure simplification rules.

In summary, the present tactic is a normal simplifier which transforms a boolean term into the simplest (symbol complexity-wise) logically equivalent term satisfying the normal form.

Before ending this section, I would like to give a second look at the two lists of rules given above. I must admit that they do not quite depict what goes on inside the simplifier. In effect, for many of the rules, the simplification tactic uses a version which is weaker than what is led to believe above. For example, it does not use (a => a) = true, but (a[1] & ... & a[n] => c[1] v ... v c[m]) = true, if a[i] = c[j] for some i and j. A similar remark applies to rules 9, 12, and 14 in the first list, and all rules in the second list except the three first ones.

But such elliptic rules are not terms of the language, neither are schemes like normalization rule 3 or simplification rule 5. If all rules were terms, it would be possible to write a very uniform simplification tactic which would specialize a rule by simple matching of its left-hand side with a subterm of the term to be simplified and replace this subterm by the instantiated right-hand side.

In fact, elliptic rules can be thought of as the application of a number of such first-order rules. Schematic rules can be dealt with by using a second-order matcher. Thus elliptic and schematic rules can be used in a natural way, and the simplification tactic can be made very uniform, let alone the fact that its consistency can then be shown more easily.

The main disadvantage of this approach is its inefficiency. Every application of an elliptic rule requires several applications of first-order rules. As for other schematic rules, I have experimented with a second-order matcher based on Huet's unification algorithm for typed lambda-calculus (1975). Such a matcher is quite complex in itself; in particular, it is non-deterministic. Unless much of its practical inefficiency can be removed, it seems to be misplaced in a simplification algorithm.

So, we have to abandon this perhaps nicer approach, and revert to writing what amounts to be little simplification subtactics for each elliptic and schematic rule. Explicitly, each of these mini-tactics are programs which test whether a term s satisfies (somehow) the left-hand side of a rule. If it does, the tactic is successful and yields the appropriately instantiated left-hand side of the rule; otherwise, the tactic fails. Figure 6-1 displays the actual POP-2 programs corresponding to the rules of (relaxed) reflexivity of => and commutativity of cond.

```
[[101 [IMPLIES] 'GENERALIZED REFLEXIVITY@]
 [LAMBDA T; VARS A C H LASTA LASTC;
  FST(T)->A; 0->LASTA;
  LOOPIF TRUE
     THEN
     IF ISVARTM(A) OR RR(A)/="AND"
        THEN 1->LASTA; A
        ELSE FST(A)
        CLOSE->H;
     SND(T)->C; 0->LASTC;
     LOOPIF TRUE
        THEN
        IF ISVARTM(C) OR RR(C)/="OR"
           THEN 1->LASTC; C
           ELSE FST(C)
           CLOSE;
        IF ==(H) THEN TRUETM,1 EXIT;
        UNLESS LASTC
           THEN SND(C)->C
           ELSE GOTO OUT
           CLOSE;
        CLOSE;
     OUT:
     UNLESS LASTA
        THEN SND(A)->A
        ELSE 0
        EXIT;
     CLOSE;
  END]]

[[211 NIL 'F(COND(A,X,Y))=COND(A,F(X),F(Y))!]
 [LAMBDA T; VARS S;
  IF ISVARTM(T) OR FSTCHAR(4,RR(T))="COND"
     OR RR(T)="IMPLIES" OR RR(T)="AND" OR RR(T)="OR"
     OR RR(T)="NOT"
     THEN 0
     EXIT;
  RD(T)->S;
  LOOPIF S/=NIL
     THEN
     UNLESS ISVARTM(HD(S)) OR FSTCHAR(4,RR(HD(S)))/="COND"
        THEN GENSYM("COND",VALUE(MEANING(TYPE(T)),"INIT")),[%
        FST(HD(S)),
        REPLACE(HD(S),SND(HD(S)),T),
        REPLACE(HD(S),TRD(HD(S)),T)%]$0,1;
        EXIT;
     TL(S)->S;
     CLOSE;
  0;
  END]]
```

Fig. 6-1. Two schematic rules or mini-tactics.

We have lost clarity to gain efficiency. Nonetheless, this is consistent with the line which has been taken so far concerning other derived rules of inference; for example, the induction rule is never explicitly instantiated before being applied: everything is done implicitly by programs. Note, however, that in the case where simplification rules are naturally terms of the language, we use the more transparent method of instantiating them by means of a first-order matcher, which is tolerably efficient.

6.3.2 Selection

The problem of selection is perhaps the most interesting one. Given a term to be simplified, we ask a double question: which subterm of the term to apply which simplification rule to? The first part of the question is very much like deciding on a computation rule as defined in chapter 3.

Before going into the heart of this matter, however, it is necessary to give some indication about the internal representation of terms in the implementation of this prover. The various algorithms given up till now have been accurate but abstract descriptions of actual programs written in POP-2. But in order to understand the present simplification algorithm, it is important to be more explicit.

The concrete syntax for terms which the user of this prover sees makes use of POP-2 lists. A variable is a word; a function application is a list whose head is a word and whose tail is a list of terms. But this is not the internal representation used by the program. There are two POP-2 functions INTOTERM and INTOLIST which

make the transformations from external to internal representations and vice versa. The printing functions do an implicit INTOLIST when they print a term.

Terms are denoted inside by three-element records. The first element is always a word; the third element is a boolean, 0 or 1. The second element is either the word UNDEF if the term is a variable, or a list of terms if the term is a function application. As it turns out, such a representation does not take up more space than lists except for variables which are not shared. This is its only disadvantage along with the fact that some non-standard and consequently slower functions had to be written for it.

On the other hand, the unusual way of representing variables has two main points in its favour. Firstly, in a term which does not involve any sharing, even of variables, dealing with term occurrences is very efficient: an occurrence is just a pointer. Since we often have to separate different occurrences of the same terms in our generalization tactics, this representation was thought to be quite advantageous. Many POP-2 functions could then be written which worked alternatively with terms (using a term equality) or term occurrences (using pointer equality). With the more common representation where variables are shared, occurrences have to be distinguished by indexing.

The second advantage of this representation has to do with simplification. This tactic can deal with variables and function applications indistinguishably, which is crucial. Once a term t has been simplified, the resulting term s, whether it is a variable or not, is copied elementwise into the record of term t. Thus any superterms which shared term t now share its simplified equivalent s.

Every opportunity of sharing subterms thus is taken in simplifying. Of course, the very last result of the simplification tactic has to be copied once in order to give any substance to the notion of associating pointers with term occurrences later.

As already hinted at in chapter 3, the answer to which subterm to select in the term to be simplified was inspired by Vuillemin's call-by-need computation rule (1973). Applied to simplification, the rule says: select the leftmost-outermost subterm which can be simplified (i.e. call-by-name), but take the maximum advantage of shared subterms.

So, on the surface of things, we first select the whole term itself, match the left of some simplification rule with it, and if this succeeds, apply the substitution found to the right of the rule: this instance is the partially simplified term. If no more rule can be applied to the outermost term and the term is an application, then we try to simplify each of the arguments. If none of them can be simplified in this way, we say that the term is fully simplified. For instance, suppose that the term to be simplified has the form (s => t) => u. Normalization rule 7, that is, (a => b) => c = (b => c) & (a v c), is applicable to it. By matching the left with our term, we get the substitution [s/a] [t/b] [u/c]; this substitution is then applied to the right which yields (t => u) & (s v u). This depicts what happens on the surface.

Internally, however, things look a bit different. Firstly, the algorithm which applies a substitution does not do undue copying, so that both occurrences of u are actually the same physical structure while the structures of t and s are left untouched. Secondly, this sharing is taken into account by the rest of the

simplification tactic. If at a later stage, u is fully simplified to u′, then u′ is copied into the physical location of u, so that every other term which shares this copy of u also benefits from its simplification. The boolean of a fully simplified term is set to 1. In other words, if the first occurrences of u is fully simplified to u′, we get (t => u′) & (s v u′) and not (t => u′) & (s v u). When the simplifier comes to the second occurrences of u′ it passes on since the boolean of u′ has been set to 1. The effect is the same as if we had first simplified u to u′, and then only applied the normalization rule 7. This brings an answer the first selection question.

The second half of the selection question concerns the order in which the various simplification rules are tried on a given term. As a first step, the rules are divided into three categories: (1) k-recursive definitions, (2) pure simplification rules, and (3) normalization rules. The difference between 2 and 3 may sometimes appear to be thin, but it is mainly a matter of convenience. Within each category, the rules are further grouped according to the function constant which is the leading left-hand side rator; all rules with a common constant on the left are listed on the property list of the constant. Ordering the rules within each of these groups is not easy. As for basic rules, it is simply the order in which they are listed above; this was chosen so as to make the greatest simplifications as soon as possible. Other rules added by the user must be specified to be of the pure simplification or normalization brand; they are simply added to the beginning of the corresponding lists.

There are in fact, two simplification tactics: (1) simplification, which uses both pure simplification rules and definitions in that order, and (2) normalization, which uses rules in all three categories. As we have seen in preceding chapters, both can be useful although the second is definitely the most important one. And this is the tactic which will be given now.

Normalization of a term t

We start by copying the main triplet of t and assigning it to a program variable s.

If the boolean of s is 1, we terminate: s is fully simplified. Before exiting, we copy each element of the main triplet of s into the corresponding element of t. (Note that variables always have their booleans set to 1.) Otherwise, we try to successively apply to s: (1) pure simplification rules, (2) k-recursive definitions, (3) normalization rules, testing after each rule whether the boolean of s is 1 or 0, until none can be applied. Then we recursively apply the present algorithm to each argument of s. If at least one can be simplified, we keep on trying to apply rules to the resulting term s; otherwise, s has been fully simplified. We terminate by copying s in t as above, setting the boolean of t to 1. ||

The decision of incorporating an efficient algorithm tactic with sharing of subterms is a very pragmatic one. It turned out to be a must in the simplification of complex terms, even if the non-standard representation takes some toll on the simplification of simpler terms. As a simple indication without statistical pretence, table 6-1 gives the time (in milliseconds) taken for the evaluation

of Ackerman's function on certain values by two different methods.

TABLE 6-1

COMPARATIVE EFFICIENCY OF TWO METHODS

	ack(2,1)	ack(2,2)	ack(3,2)
call-by-name	1549	3256	89714
call-by-need	1602	2754	56790

In the same line of thought, I promised in chapter 1 to give figures about the comparative efficiency of definitions by cases versus definitions by conditionals. In order to find the value of ack(2,1), the same call-by-need evaluator applied 67 instances of the k-recursive definition rule with ack defined by conditionals, against only 14 with ack defined by cases. Furthermore, when we do have to use conditionals in patched together definitions, the loss of efficiency of simplification is quite noticeable.

Independently of whether there is sharing or not, normalization, in particular, is very sensitive to the order in which the subterms of a term are selected. Suppose that we select leftmost innermost subterms (i.e. call-by-value) instead of leftmost outermost (i.e. call-by-name) ones. Intuitively, it seems clear that a lot of the work put into normalizing inner subterms may well have to be undone in the normalization of outer terms. Moreover, and because of this, the call-by-value rule is likely to yield a more complex normal form than call-by-name. Here is a quite telling example of what I mean: the normalization of ((a v b) => c) => d. With call-by-value, we successively get:

1. ((a v b) => c) =>d
2. ((a => c) & (b => c)) => d

3. ((c & (b => c)) => d) & ((b => c) => (a v d))
4. (c & c => d) & (c => b v d) & (c => a v d) & (a v b v d)
5. (c =>d) & (c => b v d) & (c => a v d) & (a v b v d)

whereas call-by-name yields:

1. ((a v b) => c) => d
2. (c => d) & (a v b v d);

and the gain is bigger with call-by-need since both copies of d are then shared. One can checked that the results of both normalizations are logically equivalent.

Unfortunately, with the present formal language, we cannot completely avoid doing too much work in normalization; it may happen that some inner term is put into normal form which later has to be undone. The problem comes from the fact that connectives in this language are function constants like any others and can appear at any depth in a term. This property which was felt to be desirable for attaining complete uniformity in doing induction appears to be slightly detrimental to simplification. A calculus with proper implicative, conjunctive, and disjunctive connectives on top of the present ones may do better. They could occur only as outermost symbols and normalization would only apply to them. We would also need some rules to bring to the surface all the functional connectives buried into the terms, a bit like we presently have to do with conditionals. On the other hand, this may not cause such an important problem to, say, the induction tactic, since connectives already receive a special treatment.

There are two other and more important lines of improvement which one would like to follow regarding this simplification tactic. Firstly, the complexity measure used in this prover could be extented with profit to a total order on the terms of the language; with such

a measure, one of two terms is always simpler than the other, unless they are identical. In particular, one must order constructor and function constants; as for variables, a lexical ordering seems useful. The main point about such an extension is the possibility of using the commutativity property of e.g. =, +, *, which is quite impossible at the moment. This capability is already present in many simplifiers (Brotz 1974, Moore 1974).

The other very desirable amelioration one should bring to this tactic is the possibility of using conditional equalities as simplification rules. At present, only simple equalities can be used. In general, one would like to allow equalities depending on certain conditions. More explicitly, such generalized simplification rules would be implications with an equality as unique consequent member. A subterm of a term could be simplified using the equality only if the antecedent of the rule is satisfied. This may lead to rethinking some of the tactic: the present simplifier is context-free, and what I am asking for is a context-sensitive simplifier. Such a tactic would take both a term to be simplified and a context as arguments. This context may be more easily accessible if we have outermost connectives. Other simplifiers have had a similar feature (Milner 1972, Moore 1974, Cartwright 1976). This generalized simplification would not only be useful in itself, but also in conjunction with other tactics.

CONCLUSION

In the foregoing text, I have pointed out some of the capabilities and limitations of the present methods. I have considered possible extensions, or else alternative solutions which departed from the main line of this work. I now wish to reflect on what this work has achieved and to summarize some improvements I can see for it.

I have presented a tight formal system which allowed abstract data types to be defined with a correspondingly general induction rule. The correctness proof of e.g. a compiling algorithm would be very difficult to generate without some facility for defining types. I have also introduced my tactics within a specific framework concerning search spaces and search strategies. This was quite important for justifying the consistency of the tactics.

I have also shed a new light on generalization. Since it cannot be avoided in proofs by induction, I proposed to link it directly with the selection of induction variables. Both techniques were based on a call-by-need computation rule, which allowed the generalization of only certain occurrences of a term in crucial positions.

Generalization of another sort was presented as a solution to the problem caused by non-recursion arguments which do not stay fixed. This elaborate technique was based on specialization of the goal, and on replacement in the goal with the specialized term. The purpose was to provoke the recurrence of a term on both sides of = or => in order to generalize it. The resulting variable was subject to being instantiated in the subsequent induction subgoals.

I also gave a method for generating and strengthening induction subgoals which corresponded to the general induction rule, and whose consistency was convincing enough. The method allowed for the instantiation of free variables in the induction hypotheses.

Finally, I used various other strategies: checking for non-provability of conjectures, contraction, fast simplification.

Compared with the general objective of automating the proof generation of a very large class of useful theorems, what this theorem-proving system achieves is modest. So, it may be appropriate to return to the questions asked at the beginning of this work: How much further can the Brotz and Boyer-Moore approach be pushed? How good is a pure backwark search? In my view, the present experience shows the practical limitations of subgoaling methods even more clearly. Because of the richer type structure of the language, I could tackle more difficult theorems and judge from them. One of the major problems has to do with the necessity of discarding useless antecedent and consequent members in order to keep the goals small; this was discussed in chapter 3.

Another difficulty was noted by Moore (1973): some generalizations can be achieved only if an additional hypothesis about the new variable is added. For instance,

sort(sort(l)) = sort(l)

should be generalized to

ord(k) => sort(k)=k.

Brotz's separation problem (1974) (which he solves for small problems) retains its actually for more involved ones. It concerns theorems bearing upon more than one property. For example,

(m*n)*p = m*(p*n)

involves both the commutativity and associativity of *. Brotz solves this particular instance by means of specialization.

It is my belief that problems like the three above are too hard to be solved by working from the goal only. It seems to me that the discovery of useful lemmas on a reasonably large scale will require the user's intervention (interactively or not).

Nonetheless, backwark search can give excellent hints as to which results ought to be previously proved. So, one can start by trying to prove a goal backward. This may lead to an impossible situation, but hopefully, something can be learnt about necessary lemmas. The proofs of these can then be attempted, which may or may not be obtained automatically. Finally, these proved lemmas can be used by a sophisticated simplifier in the proof of the main goal.

Clearly, backward search has a role to play. The methods developed in this thesis constitute a useful contribution to the understanding and systematization of proofs by induction, which are essentially generated top-down. Actually, their implementation has shown that even complex theorems can be proved automatically with

their help.

Apart from the general idea of more or less interactive discovery of lemmas, I have mentioned in the text more modest lines of improvement to this work. Some were even discussed to some length. I now wish to divide possible improvements into two categories according to what I believe to be their degree of difficulty.

Firstly, proper connectives could be added to the formal language without too much difficulty; this is for a pragmatic reason. The simplification algorithm could then be modified accordingly, and allow for conditional simplification rules more easily. The replacement tactic should also be reviewed, perhaps along the lines given in chapter 5. A mechanism for remembering induction subgoals which have been solved may not present too much problem to implement either.

More serious are the following questions. Techniques for handling full first-order quantification in a sufficiently controlled way are lacking at the moment; dealing with existential quantifiers may well require the synthesis of function definitions. Some form of polymorphism in the typed language seems also to be necessary. Methods for proving properties of mutually defined functions in the context of mutually defined types would be welcome. For that matter, proving properties of general recursive functions presents certainly a still greater challenge for research. One would have to prove termination as well as correctness. Substructure orderings may not be sufficient as one may need the capability of specifying the ordering of his choice and of using the corresponding induction rule easily. Computational induction is another obvious candidate for

proving properties of such programs, and it would be interesting to see to what extent the methods developed for structural induction can carry up to this other inductive method.

APPENDIX 1

TYPE AND FUNCTION DEFINITIONS

NOTE

Relevant type definitions in the syntax used by the computer program. See also appendix 3, section 3.

```
[NAT [ZERO] [SUCC NAT]]

[LIST [NIL] [CONS NAT LIST]]

[SEXPR [ATOM NAT] [CONSX SEXPR SEXPR]]

[TREE [NULLTREE]
      [TIP NAT]
      [NODE TREE NAT TREE]]

[FUNCTION]
```

NOTE

Relevant function definitions in the syntax used by the computer program. See also appendix 3, section 3.

```
[[[PRED N] NAT]
 [CASES N
        [[ZERO] [ZERO]]
        [[SUCC N] N]]]

[[[PLUS N M] NAT]
 [CASES N
        [[ZERO] M]
        [[SUCC N] [SUCC [PLUS N M]]]]]
```

```
[[[PLUS2 M N] NAT]
 [CASES M
        [[ZERO] N]
        [[SUCC M] [PLUS2 M [SUCC N]]]]]

[[[MINUS M N] NAT]
 [CASES N
        [[ZERO] M]
        [[SUCC N] [PRED [MINUS M N]]]]]

[[[MINUS2 M N] NAT]
 [CASES N
        [[ZERO] M]
        [[SUCC N] [MINUS2 [PRED M] N]]]]

[[[TIMES N M] NAT]
 [CASES N
        [[ZERO] [ZERO]]
        [[SUCC N] [PLUS M [TIMES N M]]]]]

[[[TIMES2A N M P] NAT]
 [CASES N
        [[ZERO] P]
        [[SUCC N] [TIMES2A N M [PLUS M P]]]]]

[[[TIMES2 N M] NAT]
 [TIMES2A N M [ZERO]]]

[[[EXP M N] NAT]
 [CASES N
        [[ZERO] [SUCC [ZERO]]]
        [[SUCC N] [TIMES [EXP M N] M]]]]

[[[EXP2A M N O] NAT]
 [CASES N
        [[ZERO] O]
        [[SUCC N] [EXP2A M N [TIMES M O]]]]]

[[[EXP2 M N] NAT]
 [EXP2A M N [SUCC [ZERO]]]]

[[[FACT N] NAT]
 [CASES N
        [[ZERO] [SUCC [ZERO]]]
        [[SUCC N] [TIMES [FACT N] [SUCC N]]]]]

[[[FACT2A M N] NAT]
 [CASES M
        [[ZERO] N]
        [[SUCC M]
         [FACT2A M [TIMES [SUCC M] N]]]]]

[[[FACT2 N] NAT]
 [FACT2A N [SUCC [ZERO]]]]
```

```
[[[ACK N M] NAT]
 [CASES N
        [[ZERO] [SUCC M]]
        [[SUCC N]
         [CASES M
                [[ZERO] [ACK N [SUCC [ZERO]]]]
                [[SUCC M]
                 [ACK N [ACK [SUCC N] M]]]]]]]

[[[LTE N M] BOOL]
 [CASES N
        [[ZERO] [TRUE]]
        [[SUCC N]
         [CASES M
                [[ZERO] [FALSE]]
                [[SUCC M] [LTE N M]]]]]]

[[[GT N M] BOOL]
 [CASES N
        [[ZERO] [FALSE]]
        [[SUCC N]
         [CASES M
                [[ZERO] [TRUE]]
                [[SUCC M] [GT N M]]]]]]

[[[HD L] NAT]
 [CASES L
        [[NIL] [ZERO]]
        [[CONS N L] N]]]

[[[TL L] LIST]
 [CASES L
        [[NIL] [NIL]]
        [[CONS N L] L]]]

[[[APP L K] LIST]
 [CASES L
        [[NIL] K]
        [[CONS N L] [CONS N [APP L K]]]]]

[[[LAST L] NAT]
 [CASES L
        [[NIL] [ZERO]]
        [[CONS N [NIL]] N]
        [[CONS M [CONS N L]]
         [LAST [CONS N L]]]]]

[[[LENGTH L] NAT]
 [CASES L
        [[NIL] [ZERO]]
        [[CONS N L] [SUCC [LENGTH L]]]]]
```

```
[[[REV L] LIST]
 [CASES L
        [[NIL] [NIL]]
        [[CONS N L]
         [APP [REV L] [CONS N [NIL]]]]]]

[[[REV2A L K] LIST]
 [CASES L
        [[NIL] K]
        [[CONS N L] [REV2A L [CONS N K]]]]]

[[[REV2 L] LIST] [REV2A L [NIL]]]

[[[APPLY1 F N] NAT] []]

[[[APPLY2 F M N] NAT] []]

[[[MAPLIST L F] LIST]
 [CASES L
        [[NIL] [NIL]]
        [[CONS N L]
         [CONS [APPLY1 F N] [MAPLIST L F]]]]]

[[[LIT F L N] NAT]
 [CASES L
        [[NIL] N]
        [[CONS M L]
         [APPLY2 F M [LIT F L N]]]]]

[[[ORD L] BOOL]
 [CASES L
        [[NIL] [TRUE]]
        [[CONS N [NIL]] [TRUE]]
        [[CONS M [CONS N L]]
         [AND [LTE M N] [ORD [CONS N L]]]]]]

[[[TOLIST N L] LIST]
 [CASES L
        [[NIL] [CONS N [NIL]]]
        [[CONS M L]
         [CONDL [LTE N M]
                [CONS N [CONS M L]]
                [CONS M [TOLIST N L]]]]]]

[[[SORT L] LIST]
 [CASES L
        [[NIL] [NIL]]
        [[CONS N L] [TOLIST N [SORT L]]]]]

[[[COPY S] SEXPR]
 [CASES S
        [[ATOM N] [ATOM N]]
        [[CONSX S1 S2]
         [CONSX [COPY S1] [COPY S2]]]]]
```

```
[[[FLAT S] LIST]
 [CASES S
        [[ATOM N] [CONS N [NIL]]]
        [[CONSX U V]
         [APP [FLAT U] [FLAT V]]]]]

[[[FLAT2A S L] LIST]
 [CASES S
        [[ATOM N] [CONS N L]]
        [[CONSX S1 S2]
         [FLAT2A S1 [FLAT2A S2 L]]]]]

[[[FLAT2 S] LIST]
 [CASES S
        [[ATOM N] [CONS N [NIL]]]
        [[CONSX S1 S2]
         [FLAT2A S1 [FLAT2 S2]]]]]

[[[SUBEXP T S] BOOL]
 [CASES S
        [[ATOM N] [EQS T [ATOM N]]]
        [[CONSX S1 S2]
         [OR [EQS [CONSX S1 S2] T]
             [OR [SUBEXP T S1] [SUBEXP T S2]]]]]]

[[[SUBST M N S] SEXPR]
 [CASES S
        [[ATOM P]
         [CONDS [EQN P N] [ATOM M] [ATOM P]]]
        [[CONSX S1 S2]
         [CONSX [SUBST M N S1]
                [SUBST M N S2]]]]]

[[[REVEXP S] SEXPR]
 [CASES S
        [[ATOM N] [ATOM N]]
        [[CONSX S1 S2]
         [CONSX [REVEXP S2] [REVEXP S1]]]]]

[[[SIZE S] NAT]
 [CASES S
        [[ATOM N] [SUCC [ZERO]]]
        [[CONSX S1 S2]
         [PLUS [SIZE S1] [SIZE S2]]]]]

[[[ALLLTE TR N] BOOL]
 [CASES TR
        [[NULLTREE] [TRUE]]
        [[TIP M] [LTE M N]]
        [[NODE TR1 M TR2]
         [AND [ALLLTE TR1 N] [ALLLTE TR2 N]]]]]
```

```
[[[ALLGTE TR N] BOOL]
 [CASES TR
        [[NULLTREE] [TRUE]]
        [[TIP M] [LTE N M]]
        [[NODE TR1 M TR2]
         [AND [ALLGTE TR1 N] [ALLGTE TR2 N]]]]]

[[[ORDT TR] BOOL]
 [CASES TR
        [[NULLTREE] [TRUE]]
        [[TIP N] [TRUE]]
        [[NODE TR1 N TR2]
         [AND [ORDT TR1]
              [AND [ORDT TR2]
                   [AND [ALLLTE TR1 N]
                        [ALLGTE TR2 N]]]]]]]

[[[TOTREE N TR] TREE]
 [CASES TR
        [[NULLTREE] [TIP N]]
        [[TIP M]
         [CONDT [LTE M N]
                [NODE [TIP M] M [TIP N]]
                [NODE [TIP N] M [TIP M]]]]
        [[NODE TR1 M TR2]
         [CONDT [LTE M N]
                [NODE TR1 M [TOTREE N TR2]]
                [NODE [TOTREE N TR1] M TR2]]]]]

[[[MKTREE L] TREE]
 [CASES L
        [[NIL] [NULLTREE]]
        [[CONS N L] [TOTREE N [MKTREE L]]]]]

[[[FLATTREE TR] LIST]
 [CASES TR
        [[NULLTREE] [NIL]]
        [[TIP N] [CONS N [NIL]]]
        [[NODE TR1 N TR2]
         [APP [FLATTREE TR1] [FLATTREE TR2]]]]]

[[[TREESORT L] LIST]
 [FLATTREE [MKTREE L]]]

[[[MEM N L] BOOL]
 [CASES L
        [[NIL] [FALSE]]
        [[CONS M L]
         [OR [EQN M N] [MEM N L]]]]]

[[[SUBSET K L] BOOL]
 [CASES K
        [[NIL] [TRUE]]
        [[CONS N K]
         [AND [MEM N L] [SUBSET K L]]]]]
```

```
[[[INTER K L] LIST]
 [CASES K
        [[NIL] [NIL]]
        [[CONS N K]
         [CONDL [MEM N L]
                [CONS N [INTER K L]]
                [INTER K L]]]]]

[[[INTER2A J K L] LIST]
 [CASES J
        [[NIL] L]
        [[CONS N J]
         [CONDL [MEM N K]
                [INTER2A J K [CONS N L]]
                [INTER2A J K L]]]]]

[[[INTER2 K L] LIST]
 [INTER2A K L [NIL]]]

[[[UNION K L] LIST]
 [CASES K
        [[NIL] L]
        [[CONS N K]
         [CONDL [MEM N L]
                [UNION K L]
                [CONS N [UNION K L]]]]]]

[[[UNION2 K L] LIST]
 [CASES K
        [[NIL] L]
        [[CONS N K]
         [CONDL [MEM N L]
                [UNION2 K L]
                [UNION2 K [CONS N L]]]]]]
```

APPENDIX 2

THEOREMS PROVED

In this appendix, I follow a certain tradition and give only theorems which the prover could established. The burden of finding and studying facts which the system <u>cannot</u> prove is left to the inquisitive reader. Nonetheless, I can say that at least several additional theorems could have been proved with a slightly improved replacement tactic as discussed in chapter 5. Other goals could not be achieved because of exceedingly long checkings in generalization (think of properties of functions like exp, fact, ack, or even sort); the separation problem discussed by Brotz (1974) was also a source of trouble.

```
[EQN [PLUS M N] [PLUS N M]]

[EQN [PLUS M [PLUS N O]]
     [PLUS [PLUS M N] O]]

[EQN [PRED [MINUS M N]]
     [MINUS [PRED M] N]]

[EQN [MINUS [MINUS M N] P]
     [MINUS M [PLUS P N]]]

[EQN [MINUS [MINUS M N] P]
     [MINUS [MINUS M P] N]]

[EQN [MINUS [PLUS N M] N] M]

[EQN [PLUS N [MINUS M N]]
     [PLUS M [MINUS N M]]]

[EQN [TIMES M N] [TIMES N M]]

[EQN [TIMES [TIMES M N] P]
     [TIMES M [TIMES N P]]]

[EQN [TIMES M [PLUS N P]]
     [PLUS [TIMES M N] [TIMES M P]]]

[EQN [EXP M [PLUS N P]]
     [TIMES [EXP M N] [EXP M P]]]
```

```
[GT [ACK M N] [ZERO]]

[IMPLIES [AND [LTE M N] [LTE N P]]
         [LTE M P]]

[LTE N N]

[IMPLIES [AND [LTE M N] [LTE N M]]
         [EQN M N]]

[EQB [GT M N]
     [AND [LTE N M] [NOT [LTE M N]]]]

[EQL [APP J [APP K L]]
     [APP [APP J K] L]]

[EQL [APP L [APP L L]]
     [APP [APP L L] L]]

[IMPLIES [EQL [APP K L] [APP K J]]
         [EQL L J]]

[EQN [LENGTH [APP J K]]
     [LENGTH [APP K J]]]

[EQL [REV [APP J K]]
     [APP [REV K] [REV J]]]

[EQN [LENGTH [REV L]] [LENGTH L]]

[EQL [REV [REV L]] L]

[EQN [LAST [REV L]] [HD L]]

[EQL [MAPLIST [APP K L] F]
     [APP [MAPLIST K F] [MAPLIST L F]]]

[EQN [LENGTH [MAPLIST L F]]
     [LENGTH L]]

[EQL [REV [MAPLIST L F]]
     [MAPLIST [REV L] F]]

[EQN [LIT F [APP K L] N]
     [LIT F K [LIT F L N]]]

[ORD [SORT L]]

[EQB [EQL [SORT L] L] [ORD L]]

[IMPLIES [MEM N L] [MEM N [APP L K]]]

[IMPLIES [MEM N K] [MEM N [APP L K]]]
```

```
[IMPLIES [OR [MEM N J] [MEM N K]]
         [MEM N [APP J K]]]

[IMPLIES [AND [MEM N J] [MEM N K]]
         [MEM N [INTER J K]]]

[IMPLIES [OR [MEM N J] [MEM N K]]
         [MEM N [UNION J K]]]

[IMPLIES [SUBSET K L]
         [EQL [UNION K L] L]]

[IMPLIES [SUBSET K L]
         [EQL [INTER K L] K]]

[SUBSET K K]

[EQS [COPY S] S]

[EQS [SUBST N N S] S]

[IMPLIES [NOT [SUBEXP [ATOM N] S]]
         [EQS [SUBST M N S] S]]

[EQN [LENGTH [FLAT S]] [SIZE S]]

[EQS [REVEXP [REVEXP S]] S]

[EQL [FLAT [REVEXP S]]
     [REV [FLAT S]]]
```

NOTE

I made many (vain) attempts at proving a tree sort algorithm (Burstall 1969). This is a good example where subgoaling by itself is just not good enough. I could only prove the following relevant lemmas.

```
[IMPLIES [ORD [APP K L]] [ORD K]]

[IMPLIES [ORD [APP K L]] [ORD L]]

[IMPLIES [AND [LTE M N] [ALLGTE TR M]]
         [ALLGTE [TOTREE N TR] M]]

[IMPLIES [AND [NOT [LTE M N]] [ALLLTE TR M]]
         [ALLLTE [TOTREE N TR] M]]
```

NOTE

The following theorems all involve functions with accumulators and quite a few required to be indirectly generalized. Facts about the existence of identity elements for e.g. plus, times, app, have had to be used.

```
[EQN [PLUS2 M N] [PLUS M N]]

[EQN [PLUS2 M N] [PLUS2 N M]]

[EQN [PLUS2 [PLUS2 M N] P]
     [PLUS2 M [PLUS2 N P]]]

[EQN [MINUS2 M N] [MINUS M N]]

[EQN [TIMES2 M N] [TIMES M N]]

[EQN [TIMES2 M N] [TIMES2 N M]]

[EQN [TIMES2 [TIMES2 M N] P]
     [TIMES2 M [TIMES2 N P]]]

[EQN [EXP2 M N] [EXP M N]]

[EQN [FACT2 N] [FACT N]]

[EQL [REV2 L] [REV L]]

[EQL [REV2 [REV2 L]] L]

[EQL [APP [REV2 L] [REV2 K]]
     [REV2 [APP K L]]]

[IMPLIES [MEM N L]
         [MEM N [UNION2 K L]]]

[IMPLIES [MEM N [UNION2 K L]]
         [OR [MEM N K] [MEM N L]]]

[IMPLIES [MEM N [INTER2 K L]]
         [MEM N K]]

[EQL [APP [FLAT S] L] [FLAT2A S L]]
```

APPENDIX 3

SAMPLE PROOFS

1.0 DISTRIBUTIVE LAW FOR ADDITION AND MULTIPLICATION

NOTE

Many subgoals in this proof must be generalized so as to separate different occurrences of the same subterm. This proof uses no previously proved theorems other than those of the basic amplification.

```
GOAL [ 1]
[EQN [TIMES N [PLUS N N]]
     [PLUS [TIMES N N] [TIMES N N]]]

GENERALIZATION

[EQN [TIMES NG1 [PLUS N N]]
     [PLUS [TIMES NG1 N] [TIMES NG1 N]]]

INDUCTION [NG1]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG11 / NG1 ]
```

```
GOAL [ 1 1]
[EQN [TIMES [ZERO] [PLUS N N]]
     [PLUS [TIMES [ZERO] N]
           [TIMES [ZERO] N]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2]
[IMPLIES [EQN [TIMES NG11 [PLUS N N]]
              [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
         [EQN [TIMES [SUCC NG11] [PLUS N N]]
              [PLUS [TIMES [SUCC NG11] N]
                    [TIMES [SUCC NG11] N]]]]

SIMPLIFICATION

[IMPLIES [EQN [TIMES NG11 [PLUS N N]]
              [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
         [EQN [PLUS [PLUS N N]
                    [TIMES NG11 [PLUS N N]]]
              [PLUS [PLUS N [TIMES NG11 N]]
                    [PLUS N [TIMES NG11 N]]]]]

REPLACEMENT USING HYPOTHESES

[EQN [PLUS [PLUS N N]
           [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
     [PLUS [PLUS N [TIMES NG11 N]]
           [PLUS N [TIMES NG11 N]]]]

GENERALIZATION

[EQN [PLUS [PLUS NG2 N]
           [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
     [PLUS [PLUS NG2 [TIMES NG11 N]]
           [PLUS N [TIMES NG11 N]]]]

INDUCTION [NG2]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG21 / NG2 ]

GOAL [ 1 2 1]
[EQN [PLUS [PLUS [ZERO] N]
           [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
     [PLUS [PLUS [ZERO] [TIMES NG11 N]]
           [PLUS N [TIMES NG11 N]]]]
```

```
SIMPLIFICATION

[EQN [PLUS N
           [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
     [PLUS [TIMES NG11 N]
           [PLUS N [TIMES NG11 N]]]]

GENERALIZATION

[EQN [PLUS NG3 [PLUS NG4 [TIMES NG11 N]]]
     [PLUS NG4 [PLUS NG3 [TIMES NG11 N]]]]
```

NOTE

We do a double generalization since for us NG3 and NG4 are totally equivalent.

```
INDUCTION [NG4]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG41 / NG4 ]

GOAL [ 1 2 1 1]
[EQN [PLUS NG3
           [PLUS [ZERO] [TIMES NG11 N]]]
     [PLUS [ZERO]
           [PLUS NG3 [TIMES NG11 N]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 1 2]
[IMPLIES [EQN [PLUS NG3 [PLUS NG41 [TIMES NG11 N]]]
              [PLUS NG41 [PLUS NG3 [TIMES NG11 N]]]]
         [EQN [PLUS NG3
                    [PLUS [SUCC NG41] [TIMES NG11 N]]]
              [PLUS [SUCC NG41]
                    [PLUS NG3 [TIMES NG11 N]]]]]

SIMPLIFICATION

[IMPLIES [EQN [PLUS NG3 [PLUS NG41 [TIMES NG11 N]]]
              [PLUS NG41 [PLUS NG3 [TIMES NG11 N]]]]
         [EQN [PLUS NG3
                    [SUCC [PLUS NG41 [TIMES NG11 N]]]]
              [SUCC [PLUS NG41
                          [PLUS NG3 [TIMES NG11 N]]]]]]
```

```
REPLACEMENT USING HYPOTHESES

[EQN [PLUS NG3
           [SUCC [PLUS NG41 [TIMES NG11 N]]]]
     [SUCC [PLUS NG3
                 [PLUS NG41 [TIMES NG11 N]]]]]

INDUCTION [NG3]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG31 / NG3 ]

GOAL [ 1 2 1 2 1]
[EQN [PLUS [ZERO]
           [SUCC [PLUS NG41 [TIMES NG11 N]]]]
     [SUCC [PLUS [ZERO]
                 [PLUS NG41 [TIMES NG11 N]]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 1 2 2]
[IMPLIES [EQN [PLUS NG31
                    [SUCC [PLUS NG41 [TIMES NG11 N]]]]
              [SUCC [PLUS NG31
                          [PLUS NG41 [TIMES NG11 N]]]]]
         [EQN [PLUS [SUCC NG31]
                    [SUCC [PLUS NG41 [TIMES NG11 N]]]]
              [SUCC [PLUS [SUCC NG31]
                          [PLUS NG41 [TIMES NG11 N]]]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 2]
[IMPLIES [EQN [PLUS [PLUS NG21 N]
                    [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
              [PLUS [PLUS NG21 [TIMES NG11 N]]
                    [PLUS N [TIMES NG11 N]]]]
         [EQN [PLUS [PLUS [SUCC NG21] N]
                    [PLUS [TIMES NG11 N] [TIMES NG11 N]]]
              [PLUS [PLUS [SUCC NG21] [TIMES NG11 N]]
                    [PLUS N [TIMES NG11 N]]]]]

SIMPLIFICATION

[TRUE] 131.386 SEC.
```

2.0 ASSOCIATIVITY OF MULTIPLICATION WITH ACCUMULATOR

NOTE

Multiplication is defined with an accumulator and consequently several subgoals in this proof must be indirectly generalized. This proof does not use any previously proved theorems either.

```
GOAL [ 1]
[EQN [TIMES2 [TIMES2 M N] P]
     [TIMES2 M [TIMES2 N P]]]

SIMPLIFICATION

[EQN [TIMES2A [TIMES2A M N [ZERO]
              P
              [ZERO]]
     [TIMES2A M
              [TIMES2A N P [ZERO]
              [ZERO]]]

GENERALIZATION

[EQN [TIMES2A [TIMES2A M N NG2] P [ZERO]]
     [TIMES2A M
              [TIMES2A N P [ZERO]]
              [TIMES2A NG2 P [ZERO]]]]

INDUCTION [M]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ M1 / M ] [ [PLUS N NG2] / NG2 ]

GOAL [ 1 1]
[EQN [TIMES2A [TIMES2A [ZERO] N NG2]
              P
              [ZERO]]
     [TIMES2A [ZERO]
              [TIMES2A N P [ZERO]]
              [TIMES2A NG2 P [ZERO]]]]

SIMPLIFICATION

[TRUE]
```

```
GOAL [ 1 2]
[IMPLIES [EQN [TIMES2A [TIMES2A M1 N [PLUS N NG2]]
                       P
                       [ZERO]]
              [TIMES2A M1
                       [TIMES2A N P [ZERO]]
                       [TIMES2A [PLUS N NG2] P [ZERO]]]]
         [EQN [TIMES2A [TIMES2A [SUCC M1] N NG2]
                       P
                       [ZERO]]
              [TIMES2A [SUCC M1]
                       [TIMES2A N P [ZERO]]
                       [TIMES2A NG2 P [ZERO]]]]]

SIMPLIFICATION

[IMPLIES [EQN [TIMES2A [TIMES2A M1 N [PLUS N NG2]]
                       P
                       [ZERO]]
              [TIMES2A M1
                       [TIMES2A N P [ZERO]]
                       [TIMES2A [PLUS N NG2] P [ZERO]]]]
         [EQN [TIMES2A [TIMES2A M1 N [PLUS N NG2]]
                       P
                       [ZERO]]
              [TIMES2A M1
                       [TIMES2A N P [ZERO]]
                       [PLUS [TIMES2A N P [ZERO]]
                             [TIMES2A NG2 P [ZERO]]]]]]

REPLACEMENT USING HYPOTHESES

[EQN [TIMES2A M1
              [TIMES2A N P [ZERO]]
              [TIMES2A [PLUS N NG2] P [ZERO]]]
     [TIMES2A M1
              [TIMES2A N P [ZERO]]
              [PLUS [TIMES2A N P [ZERO]]
                    [TIMES2A NG2 P [ZERO]]]]]

CONTRACTION

[EQN [TIMES2A [PLUS N NG2] P [ZERO]]
     [PLUS [TIMES2A N P [ZERO]]
           [TIMES2A NG2 P [ZERO]]]]

GENERALIZATION

[EQN [TIMES2A [PLUS N NG2]
              P
              [PLUS NG4 [ZERO]]]
     [PLUS [TIMES2A N P NG4]
           [TIMES2A NG2 P [ZERO]]]]
```

```
INDUCTION [N]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ N1 / N ] [ [PLUS P NG4] / NG4 ]

GOAL [ 1 2 1]
[EQN [TIMES2A [PLUS [ZERO] NG2]
              P
              [PLUS NG4 [ZERO]]]
     [PLUS [TIMES2A [ZERO] P NG4]
           [TIMES2A NG2 P [ZERO]]]]

SIMPLIFICATION

[EQN [TIMES2A NG2 P [PLUS NG4 [ZERO]]]
     [PLUS NG4 [TIMES2A NG2 P [ZERO]]]]

INDUCTION [NG4]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG41 / NG4 ]

GOAL [ 1 2 1 1]
[EQN [TIMES2A NG2 P [PLUS [ZERO] [ZERO]]]
     [PLUS [ZERO] [TIMES2A NG2 P [ZERO]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 1 2]
[IMPLIES [EQN [TIMES2A NG2 P [PLUS NG41 [ZERO]]]
              [PLUS NG41 [TIMES2A NG2 P [ZERO]]]]
         [EQN [TIMES2A NG2
                       P
                       [PLUS [SUCC NG41] [ZERO]]]
              [PLUS [SUCC NG41]
                    [TIMES2A NG2 P [ZERO]]]]]

SIMPLIFICATION

[IMPLIES [EQN [TIMES2A NG2 P [PLUS NG41 [ZERO]]]
              [PLUS NG41 [TIMES2A NG2 P [ZERO]]]]
         [EQN [TIMES2A NG2
                       P
                       [SUCC [PLUS NG41 [ZERO]]]]
              [SUCC [PLUS NG41 [TIMES2A NG2 P [ZERO]]]]]]
```

```
REPLACEMENT USING HYPOTHESES

[EQN [TIMES2A NG2
              P
              [SUCC [PLUS NG41 [ZERO]]]]
     [SUCC [TIMES2A NG2 P [PLUS NG41 [ZERO]]]]]

GENERALIZATION

[EQN [TIMES2A NG2 P [SUCC NG8]]
     [SUCC [TIMES2A NG2 P NG8]]]

INDUCTION [NG2]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ NG21 / NG2 ] [ [PLUS P NG8] / NG8 ]

GOAL [ 1 2 1 2 1]
[EQN [TIMES2A [ZERO] P [SUCC NG8]]
     [SUCC [TIMES2A [ZERO] P NG8]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 1 2 2]
[IMPLIES [EQN [TIMES2A NG21 P [SUCC [PLUS P NG8]]]
              [SUCC [TIMES2A NG21 P [PLUS P NG8]]]]
         [EQN [TIMES2A [SUCC NG21] P [SUCC NG8]]
              [SUCC [TIMES2A [SUCC NG21] P NG8]]]]

SIMPLIFICATION

[IMPLIES [EQN [TIMES2A NG21 P [SUCC [PLUS P NG8]]]
              [SUCC [TIMES2A NG21 P [PLUS P NG8]]]]
         [EQN [TIMES2A NG21 P [PLUS P [SUCC NG8]]]
              [SUCC [TIMES2A NG21 P [PLUS P NG8]]]]]

REPLACEMENT USING HYPOTHESES

[EQN [TIMES2A NG21 P [PLUS P [SUCC NG8]]]
     [TIMES2A NG21 P [SUCC [PLUS P NG8]]]]

CONTRACTION

[EQN [PLUS P [SUCC NG8]]
     [SUCC [PLUS P NG8]]]
```

```
INDUCTION [P]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ P1 / P ]

GOAL [ 1 2 1 2 2 1]
[EQN [PLUS [ZERO] [SUCC NG8]]
     [SUCC [PLUS [ZERO] NG8]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 1 2 2 2]
[IMPLIES [EQN [PLUS P1 [SUCC NG8]]
              [SUCC [PLUS P1 NG8]]]
         [EQN [PLUS [SUCC P1] [SUCC NG8]]
              [SUCC [PLUS [SUCC P1] NG8]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 2]
[IMPLIES [EQN [TIMES2A [PLUS N1 NG2]
                       P
                       [PLUS [PLUS P NG4] [ZERO]]]
              [PLUS [TIMES2A N1 P [PLUS P NG4]]
                    [TIMES2A NG2 P [ZERO]]]]
         [EQN [TIMES2A [PLUS [SUCC N1] NG2]
                       P
                       [PLUS NG4 [ZERO]]]
              [PLUS [TIMES2A [SUCC N1] P NG4]
                    [TIMES2A NG2 P [ZERO]]]]]

SIMPLIFICATION

[IMPLIES [EQN [TIMES2A [PLUS N1 NG2]
                       P
                       [PLUS [PLUS P NG4] [ZERO]]]
              [PLUS [TIMES2A N1 P [PLUS P NG4]]
                    [TIMES2A NG2 P [ZERO]]]]
         [EQN [TIMES2A [PLUS N1 NG2]
                       P
                       [PLUS P [PLUS NG4 [ZERO]]]]
              [PLUS [TIMES2A N1 P [PLUS P NG4]]
                    [TIMES2A NG2 P [ZERO]]]]]
```

```
REPLACEMENT USING HYPOTHESES

[EQN [TIMES2A [PLUS N1 NG2]
              P
              [PLUS P [PLUS NG4 [ZERO]]]]
     [TIMES2A [PLUS N1 NG2]
              P
              [PLUS [PLUS P NG4] [ZERO]]]]

CONTRACTION

[EQN [PLUS P [PLUS NG4 [ZERO]]]
     [PLUS [PLUS P NG4] [ZERO]]]

INDUCTION [P]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ P1 / P ]

GOAL [ 1 2 2 1]
[EQN [PLUS [ZERO] [PLUS NG4 [ZERO]]]
     [PLUS [PLUS [ZERO] NG4] [ZERO]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2 2 2]
[IMPLIES [EQN [PLUS P1 [PLUS NG4 [ZERO]]]
              [PLUS [PLUS P1 NG4] [ZERO]]]
         [EQN [PLUS [SUCC P1] [PLUS NG4 [ZERO]]]
              [PLUS [PLUS [SUCC P1] NG4] [ZERO]]]]

SIMPLIFICATION

[TRUE] 226.804 SEC.
```

3.0 CORRECTNESS OF COMPILING ALGORITHM FOR EXPRESSIONS

NOTE

This proof shows how type definitions can be useful in a complex situation. The use of vacuously defined type and defined function constants make the proof more abstract.

NOTE

We start by defining the syntax of the source language of expressions by means of type expressions.

```
[NAME]

[OPERATOR]

[EXPRESS [SIMPLE NAME] [COMPOUND OPERATOR EXPRESS EXPRESS]]
```

NOTE

Type defintions are also used for semantic domains. States associate numbers to names.

```
[FUNCTION]

[NAT [ZERO] [SUCC NAT]]

[STATE]
```

NOTE

The following semantic functions give the meaning of syntactic constructs, and in particular, MSE can be called an interpreter.

```
[[[APPLY F M N] NAT] []]

[[[MO OP] FUNCTION] []]

[[[LOOKUP NM ST] NAT] []]

[[[MSE E ST] NAT]
 [CASES E
        [[SIMPLE NM] [LOOKUP NM ST]]
        [[COMPOUND OP E1 E2]
         [APPLY [MO OP] [MSE E1 ST] [MSE E2 ST]]]]]
```

NOTE

The target language is a set of programs, which are lists of instructions. Postfixed notation is used.

```
[INSTRUCT [OPERATE OPERATOR] [FETCH NAME]]

[PROGRAM [NULLPR] [ADD INSTRUCT PROGRAM]]
```

NOTE

The semantic domains of the target language are: a pushdown (or stack) and a store. Here, we find the type state introduced above.

```
[PUSHDOWN [EMPTY] [PUSH NAT PUSHDOWN]]

[STORE [MKSTORE STATE PUSHDOWN]]
```

NOTE

We need selectors for manipulating pushdowns and stores. Popping the stack is achieved by taking off the TOP of the stack, and replacing the stack by the BOTTOM of the stack.

```
[[[TOP PD] NAT]
 [CASES PD
        [[EMPTY] [ZERO]]
        [[PUSH N PD] N]]]

[[[BOTTOM PD] PUSHDOWN]
 [CASES PD
        [[EMPTY] [EMPTY]]
        [[PUSH N PD] PD]]]

[[[STOF STR] STATE]
 [CASES STR [[MKSTORE ST PD] ST]]]

[[[PDOF STR] PUSHDOWN]
 [CASES STR [[MKSTORE ST PD] PD]]]
```

NOTE

Now, the following semantic functions execute a target language program.

```
[[[DO IN STR] STORE]
 [CASES IN
        [[FETCH NM]
         [MKSTORE [STOF STR]
                  [PUSH [LOOKUP NM [STOF STR]]
                        [PDOF STR]]]]
        [[OPERATE OP]
         [MKSTORE [STOF STR]
                  [PUSH [APPLY [MO OP]
                               [TOP [BOTTOM [PDOF STR]]]
                               [TOP [PDOF STR]]]
                        [BOTTOM [BOTTOM [PDOF STR]]]]]]]]

[[[MT PR STR] STORE]
 [CASES PR
        [[NULLPR] STR]
        [[ADD IN PR] [MT PR [DO IN STR]]]]]
```

NOTE

Finally, the function COMP compiles an expression, that is, it translates it into a program. The other function is used to concatenate programs together.

```
[[[APPROG PR1 PR2] PROGRAM]
 [CASES PR1
        [[NULLPR] PR2]
        [[ADD IN PR1]
         [ADD IN [APPROG PR1 PR2]]]]]

[[[COMP E] PROGRAM]
 [CASES E
        [[SIMPLE NM] [ADD [FETCH NM] [NULLPR]]]
        [[COMPOUND OP E1 E2]
         [APPROG [COMP E1]
                 [APPROG [COMP E2]
                         [ADD [OPERATE OP] [NULLPR]]]]]]]
```

NOTE

We first need to prove a lemma and add it to the set of normalization rules. This can in fact be viewed as an instance of a more general fact: f(app(k,l),x) = f(l,f(k,x)), where the second argument of f is an accumulator which does not depend on the first.

```
GOAL [ 1]
[EQST [MT [APPROG PR1 PR2] STR]
      [MT PR2 [MT PR1 STR]]]

INDUCTION [PR1]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ PR11 / PR1 ] [ [DO I1 STR] / STR ]

GOAL [ 1 1]
[EQST [MT [APPROG [NULLPR] PR2] STR]
      [MT PR2 [MT [NULLPR] STR]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2]
[IMPLIES [EQST [MT [APPROG PR11 PR2] [DO I1 STR]]
               [MT PR2 [MT PR11 [DO I1 STR]]]]
         [EQST [MT [APPROG [ADD I1 PR11] PR2] STR]
               [MT PR2 [MT [ADD I1 PR11] STR]]]]

SIMPLIFICATION

[TRUE] 8.015 SEC.
```

NOTE

The main theorem can now be proved. It says that compiling an expression and executing the resulting program leaves on the stack the same number as interpreting the expression and pushing the result on the stack; moreover, the state and the bottom of the stack are left unchanged.

```
GOAL [ 1]
[EQST [MT [COMP E] STR]
      [MKSTORE [STOF STR]
               [PUSH [MSE E [STOF STR]] [PDOF STR]]]]

INDUCTION [E]

SUBSTITUTIONS IN HYPOTHESES
[ 1]
NIL
[ 2]
[ E2 / E ] [ [MT [COMP E1] STR] / STR ]
[ E1 / E ]

GOAL [ 1 1]
[EQST [MT [COMP [SIMPLE N1]] STR]
      [MKSTORE [STOF STR]
               [PUSH [MSE [SIMPLE N1] [STOF STR]]
                     [PDOF STR]]]]

SIMPLIFICATION

[TRUE]

GOAL [ 1 2]
[IMPLIES [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
                    [MKSTORE [STOF [MT [COMP E1] STR]]
                             [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]]
                                   [PDOF [MT [COMP E1] STR]]]]]
              [EQST [MT [COMP E1] STR]
                    [MKSTORE [STOF STR]
                             [PUSH [MSE E1 [STOF STR]]
                                   [PDOF STR]]]]]
         [EQST [MT [COMP [COMPOUND O1 E1 E2]] STR]
               [MKSTORE [STOF STR]
                        [PUSH [MSE [COMPOUND O1 E1 E2] [STOF STR]]
                              [PDOF STR]]]]]
```

SIMPLIFICATION

```
[AND
 [IMPLIES
 .       [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
 .       .          [MKSTORE [STOF [MT [COMP E1] STR]]
 .       .                   [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]
 .       .                         [PDOF [MT [COMP E1] STR]]]]]
 .       .    [EQST [MT [COMP E1] STR]
 .       .          [MKSTORE [STOF STR]
 .       .                   [PUSH [MSE E1 [STOF STR]]
 .       .                         [PDOF STR]]]]]
 .       [EQS [STOF [MT [COMP E2] [MT [COMP E1] STR]]
 .            [STOF STR]]]
 [AND
  [IMPLIES
  .[AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
  .           [MKSTORE [STOF [MT [COMP E1] STR]]
  .                    [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]
  .                          [PDOF [MT [COMP E1] STR]]]]]
  .     [EQST [MT [COMP E1] STR]
  .           [MKSTORE [STOF STR]
  .                    [PUSH [MSE E1 [STOF STR]]
  .                          [PDOF STR]]]]]
  .[EQNA
  .    [APPLY [MO O1]
  .    .      [TOP [BOTTOM [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
  .    .      [TOP [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
  .    [APPLY [MO O1]
  .           [MSE E1 [STOF STR]]
  .           [MSE E2 [STOF STR]]]]]
  [IMPLIES
     [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
     .          [MKSTORE [STOF [MT [COMP E1] STR]]
     .                   [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]
     .                         [PDOF [MT [COMP E1] STR]]]]]
     .    [EQST [MT [COMP E1] STR]
     .          [MKSTORE [STOF STR]
     .                   [PUSH [MSE E1 [STOF STR]]
     .                         [PDOF STR]]]]]
     [EQPU [BOTTOM [BOTTOM [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
           [PDOF STR]]]]]
```

SPLITTING

```
GOAL [ 1 2 1]
[IMPLIES [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
                    [MKSTORE [STOF [MT [COMP E1] STR]]
                             [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]]
                                   [PDOF [MT [COMP E1] STR]]]]]
              [EQST [MT [COMP E1] STR]
                    [MKSTORE [STOF STR]
                             [PUSH [MSE E1 [STOF STR]]
                                   [PDOF STR]]]]]
         [EQS [STOF [MT [COMP E2] [MT [COMP E1] STR]]]
              [STOF STR]]]
```

REPLACEMENT USING HYPOTHESES

```
[EQS
   [STOF [MKSTORE [STOF [MKSTORE [STOF STR]
   .                             [PUSH [MSE E1 [STOF STR]]
   .                                   [PDOF STR]]]]
   .              [PUSH [MSE E2
   .                         [STOF [MKSTORE [STOF STR]
   .                                        [PUSH [MSE E1 [STOF STR]]
   .                                              [PDOF STR]]]]]
   .                    [PDOF [MKSTORE [STOF STR]
   .                                   [PUSH [MSE E1 [STOF STR]]
   .                                         [PDOF STR]]]]]]]
   [STOF STR]]
```

SIMPLIFICATION

```
[TRUE]
```

```
GOAL [ 1 2 2]
[IMPLIES
 [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
 .          [MKSTORE [STOF [MT [COMP E1] STR]]
 .                   [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]]
 .                         [PDOF [MT [COMP E1] STR]]]]]
 .    [EQST [MT [COMP E1] STR]
 .          [MKSTORE [STOF STR]
 .                   [PUSH [MSE E1 [STOF STR]]
 .                         [PDOF STR]]]]]
 [EQNA [APPLY [MO O1]
              [TOP [BOTTOM [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
              [TOP [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
       [APPLY [MO O1]
              [MSE E1 [STOF STR]]
              [MSE E2 [STOF STR]]]]]
```

REPLACEMENT USING HYPOTHESES

```
[EQNA
 [APPLY
 .[MO O1]
 .[TOP
 . [BOTTOM
 .  [PDOF
 .       [MKSTORE [STOF [MKSTORE [STOF STR]
 .                               [PUSH [MSE E1 [STOF STR]]
 .                                     [PDOF STR]]]]
 .                [PUSH [MSE E2
 .                           [STOF [MKSTORE [STOF STR]
 .                                          [PUSH [MSE E1 [STOF STR]]
 .                                                [PDOF STR]]]]]
 .                      [PDOF [MKSTORE [STOF STR]
 .                                     [PUSH [MSE E1 [STOF STR]]
 .                                           [PDOF STR]]]]]]]]]
 .[TOP
 . [PDOF [MKSTORE [STOF [MKSTORE [STOF STR]
 .                               [PUSH [MSE E1 [STOF STR]]
 .                                     [PDOF STR]]]]
 .                [PUSH [MSE E2
 .                           [STOF [MKSTORE [STOF STR]
 .                                          [PUSH [MSE E1 [STOF STR]]
 .                                                [PDOF STR]]]]]
 .                      [PDOF [MKSTORE [STOF STR]
 .                                     [PUSH [MSE E1 [STOF STR]]
 .                                           [PDOF STR]]]]]]]]]
 [APPLY [MO O1]
        [MSE E1 [STOF STR]]
        [MSE E2 [STOF STR]]]]
```

SIMPLIFICATION

[TRUE]

```
GOAL [ 1 2 3]
[IMPLIES
     [AND [EQST [MT [COMP E2] [MT [COMP E1] STR]]
     .          [MKSTORE [STOF [MT [COMP E1] STR]]
     .                   [PUSH [MSE E2 [STOF [MT [COMP E1] STR]]
     .                         [PDOF [MT [COMP E1] STR]]]]]
     .    [EQST [MT [COMP E1] STR]
     .          [MKSTORE [STOF STR]
     .                   [PUSH [MSE E1 [STOF STR]]
     .                         [PDOF STR]]]]].
     [EQPU [BOTTOM [BOTTOM [PDOF [MT [COMP E2] [MT [COMP E1] STR]]]]]
           [PDOF STR]]]
```

REPLACEMENT USING HYPOTHESES

```
[EQPU
 [BOTTOM
 .[BOTTOM
 . [PDOF [MKSTORE [STOF [MKSTORE [STOF STR]
 .                              [PUSH [MSE E1 [STOF STR]]
 .                                    [PDOF STR]]]]
 .                [PUSH [MSE E2
 .                          [STOF [MKSTORE [STOF STR]
 .                                         [PUSH [MSE E1 [STOF STR]]
 .                                               [PDOF STR]]]]]
 .                      [PDOF [MKSTORE [STOF STR]
 .                                    [PUSH [MSE E1 [STOF STR]]
 .                                          [PDOF STR]]]]]]]]]
 [PDOF STR]]
```

SIMPLIFICATION

[TRUE] 24.941 SEC.

APPENDIX 4

NOTE ON IMPLEMENTATION

As hinted at here and there in the dissertation, the present prover is implemented in POP-2 (Burstall, Collins, and Popplestone 1971). This programming language makes list processing easy and its general record facility allowed the representation of terms as triplets in a natural way (see chapter 6). On top of POP-2, I made extensive use of the library program LBASE written by Harry Barrow which offers general functions for manipulating property lists. A pretty-print function by Boyer (1973) was also used.

The program text adds up to roughly 4500 lines of formatted and commented POP-2 code, or alternatively occupies about 175 blocks on disc. The algorithms spelled out in the dissertation are faithfull transliterations of the main POP-2 functions of the program. Other functions are of a general nature or are used for input-output purposes. The program runs on a DECsystem10 with KA10 processor and the compiled version occupies 33K of core on top of the sharable 11K of the POP-2 system. This covers the basic amplification; type and function definitions have to be added. Nonetheless, because of the compact representation of the search space, relatively little extra store is needed in the course of generating proofs, so that most of them can be carried out without

exceeding 50K.

One might have been puzzled over the disparity of time taken by the proofs displayed in the preceding appendix; for example, the associativity of multiplication takes up 225 seconds, while both parts of the correctness proof of the compiling algorithm use up only 8 and 25 seconds respectively. The associativity proof involves three generalizations and three contractions, each of which must be checked. As pointed out in chapter 2, searching for counter-examples is quite time consuming. The figures given above are somewhat typical of the time taken by problems involving a more or less heavy use of the checker.

BIBLIOGRAPHY

Aubin, Raymond. "Some generalization heuristics in proofs by induction." In Actes du colloque Construction, amélioration et vérification de programmes, pp. 197-208. Edité par G. Huet et G. Kahn. Rocquencourt, France: Institut de recherche d'informatique et d'automatique, 1975.

Bledsoe, W. W. "Splitting and reduction heuristics in automatic theorem proving." Art. Intell. 3, 1 (Spring 1971): 55-77.

Boyer, Robert S. "Pretty-print." Memo no 64, Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, 1973.

Boyer, Robert S., and Moore, J. "Proving theorems about LISP functions." JACM 22, 1 (January 1975): 129-144.

Brotz, Douglas K. "Embedding heuristic problem solving methods in a mechanical theorem prover." STAN-CS-74-443, Computer Science Department, Stanford University, 1974.

Brown, F. Malloy. "A deductive system for elementary arithmetic." In Proc. of AISB Summer Conference, pp. 84-93. Edinburgh: The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1976.

Brown, F. Malloy. "The role of extensible deductive systems in mathematical reasoning". In Proc. of AISB Summer Conference, pp. 74-83. Edinburgh: The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1976.

Burstall, R. M. "An alternative view of J Moore's generalisation technique." Private communication, 1974.

Burstall, R. M. "Proving properties of programs by structural induction." Computer Journal 12, 1 (February 1969): 41-48.

Burstall, R. M.; Collins, J. S.; and Popplestone, R. J. Programming in POP-2. Edinburgh: University Press, 1971.

Burstall, R. M., and Darlington, John. "A transformation system for developing recursive programs." DAI Research Report no 19, Department of Artificial Intelligence, University of Edinburgh, 1976.

Cartwright, Robert. "User-defined data types as an aid to verifying LISP programs." In Proc. of Third International Colloquium on Automata, Languages, and Programming, pp. 228-256. Edited by S. Michaelson and R. Milner. Edinburgh: University Press, 1976.

Cohn, P. M. Universal algebra. New-York: Harper Row, 1965. Tokyo: Weatherhill, 1965.

Cooper, D. C. "The equivalence of certain computations." Center for the Study of Information Processing, Carnegie Institute of Technology.

Dahl, O.-J.; Dijkstra, E. W.; and Hoare, C. A. R. Structured Programming. London: Academic Press, 1972.

Darlington, Jared L. "Automatic theorem proving with equality substitutions and mathematical induction." In Machine Intelligence 3, pp. 113-127. Edited by Donald Michie. Edinburgh: University Press, 1968.

Darlington, John, and Burstall, R. M. "A system which automatically improves programs." Acta Informatica 6 (1976): 41-60.

Gerlernter, H. "Realization of a geometry-theorem proving machine." In Computers and thought, pp. 134-152. Edited by E. Feigenbaum and J. Feldman. New-York: McGraw-Hill, 1963.

Gentzen, Gerhard. Recherches sur la déduction logique. Traduit et commenté par R. Feys et J. Ladrière. Paris: Presses universitaires de France, 1955.

Gordon, M.; Milner, R.; and Wadsworth, C. "The LCF manual," Department of Computer Science, University of Edinburgh, 1976.

Henke, Friedrich W. von. "On automating proofs by induction." Private communication, 1975.

Hoare, C. A. R. "Recursive data structures." Memo AIM-223 / STAN-CS-72-400, Computer Science Department, Stanford University, 1973.

Huet, G. P. "A unification algorithm for typed lambda-calculus." Theoritical Computer Science, 1, 1 (June 1975): 27-57.

Katz, Shmuel M., and Manna, Zohar. "A heuristic approach to program verification." In Proc. of Third International Conference on Artificial Intelligence, pp. 500-512. Stanford: International Joint Council on Artificial Intelligence, 1973.

Ketonen, Oiva. "Untersuchungen zum pradikatenkalkul." Reviewed by P. Bernays. Journal of Symbolic Logic 10, 4 (December 1945): 127-130.

Kleene, Stephen Cole. Introduction to metamathematics. Gronigen: Wolters-Noordhoff, 1971.

Newey, Malcolm. "Formal semantics of LISP with applications to program correctness." Memo AIM-257 / STAN-CS-75-475, Computer Science Department, Stanford University, 1975.

Peter, Rozsa. *Recursive functions*. New-York: Academic Press, 1967.

Polya, George. *Mathematical Discovery*. New-York: Wiley, 1965.

Prawitz, Dag. "Ideas and results of proof theory." In *Proc. of Second Scandinavian Logic Symposium*, pp. 235-307. Edited by J. E. Fenstad. Amsterdam: North-Holland, 1971.

Preparata, Franco P., and Yeh, Raymond T. *Introduction to discrete structures*. Reading, Mass.: Addison-Wesley, 1973.

Robbin, Joel W. *Mathematical logic*. New-York: Benjamin, 1969.

Topor, Rodney. "Interactive program verification using virtual programs." Ph.D. thesis, University of Edinburgh, 1975.

Vuillemin, Jean E. "Proof techniques for recursive programs." Memo AIM-218 / STAN-CS-73-393, Computer Science Department, Stanford University, 1973.

Wang, Hao. "Toward mechanical mathematics." *IBM Journal of Research and Development* 4, 1 (January 1960): 2-22.