

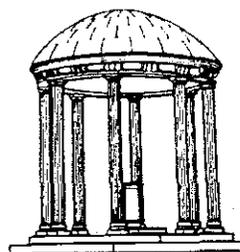
Applications of Unskolemization

TR91-027

June, 1991

Ritu Chadha

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

Applications of Unskolemization

by

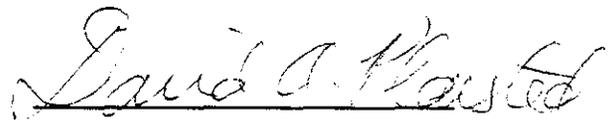
Ritu Chadha

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1991

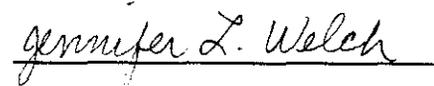
Approved by:



Advisor : David A. Plaisted



Reader : Jan F. Prins



Reader : Jennifer L. Welch

©1991
Ritu Chadha
ALL RIGHTS RESERVED

RITU CHADHA. Applications of Unskolemization
(Under the direction of David A. Plaisted)

ABSTRACT

This dissertation describes a novel method for deriving logical consequences of first-order formulas using resolution and unskolemization. A complete unskolemization algorithm is given and its properties analyzed. This method is then applied to a number of different fields, namely program verification, machine learning, and mathematical induction.

The foremost problem in automating program verification is the difficulty of mechanizing the generation of inductive assertions for loops in a program. We show that this problem can be viewed as one of generating logical consequences of the conditions which are true at the entry to a loop. A complete and sound algorithm for generating loop invariants in first-order logic is described. All previous techniques given in the literature for deriving loop invariants are heuristic in nature and are not complete in any sense.

There are a number of problems associated with machine learning, such as the diversity of representation languages used and the complexity of learning. We present a graph-based polynomial-time algorithm for learning from examples which makes use of the method for generating logical consequences. The representation language used is first-order logic, which enables the algorithm to be applied in a large number of fields where first-order logic is the language of choice. The algorithm is shown to compare favorably with others in the literature, and applications of the algorithm in a number of fields are demonstrated.

The difficulty of mechanizing mathematical induction in existing theorem provers is due to the inexpressibility of the principle of induction in first-order logic. In order to handle mathematical induction within the framework of first-order logic, it is necessary to find an induction schema for each theorem. We describe two methods for tackling this problem, one of which makes use of our method for generating logical consequences. Most existing methods for mechanizing induction can only handle equational theorems. Our approach is more general and is applicable to equational as well as non-equational theorems.

ACKNOWLEDGEMENTS

At the outset, I would like to express my profound gratitude to Professor David Plaisted for his guidance, suggestions and help throughout the duration of this research. In spite of his busy schedule, he always found time to discuss my problems and ideas and was an inexhaustible source of advice and interesting suggestions. I consider myself very lucky to have had the opportunity to work with him.

I would also like to thank the other members of my committee; Professor Jennifer Welch, for going through my thesis so painstakingly and for spending so much time with me; Professor Don Stanat, for his words of praise and encouragement and all his valuable advice, which I will cherish; Professor Alan Biermann, for the trouble he took to travel to Chapel Hill from Duke University each time I scheduled a committee meeting, and for his advice and encouragement; and Professor Jan Prins, for his critical appraisal of the program verification work and for his suggestions. Thanks are also due to fellow graduate students Heng Chu, Geoff Alexander, Suresh Rajgopal and Yong-Mi Kim for taking the time to help me with my final presentation; to Katrina Coble, who always had all the answers to my questions about the innumerable hurdles and stumbling blocks on the long road to graduation; and to everyone in the department of Computer Science for making the three years which I spent here so enjoyable.

I am grateful to the National Science Foundation for providing support for this work under Grant CCR-8802282.

Finally, I would like to thank my parents and sisters for their love and support and their unfailing confidence in me; and my husband Arindam Datta for being a constant source of cheer, support, advice, and encouragement, and for his willingness to endure a long-distance relationship during my stay at Chapel Hill.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
1.1 A brief introduction to first-order logic	2
1.1.1 Preliminaries	2
1.1.2 Syntax	3
1.1.3 Semantics	5
1.1.4 Normal forms and related definitions	7
1.1.5 Proof procedures	11
1.1.6 Theories	14
1.2 Outline of this dissertation	15
II. FINDING LOGICAL CONSEQUENCES USING UNSKOLEMIZATION	17
2.1 Objective and motivation	17
2.2 The unskolemization process	18
2.2.1 Preliminaries	18
2.2.2 The unskolemization algorithm	22
2.3 Analysis of the unskolemization algorithm	24
2.4 Summary	45
III. MECHANICAL GENERATION OF LOOP INVARIANTS FOR PROGRAM VERIFICATION	46
3.1 Past work	46
3.2 Floyd's inductive assertions method	57
3.3 Overview of the method	59
3.4 Some observations about the programming language model	61
3.5 Description of algorithm for generating loop invariants	62
3.6 The iteration algorithm	64
3.7 The function GET-APPROX	65
3.8 Proof of completeness of the iteration algorithm	70
3.9 Proof of soundness of the iteration algorithm	74
3.10 A refinement	75
3.11 Some examples	76

IV. LEARNING FROM EXAMPLES IN FIRST-ORDER LOGIC	. 93
4.1 Introduction	. 93
4.2 Motivation	. 101
4.3 Role of bias in learning	. 103
4.4 A method for learning from examples	. 105
4.4.1 Definitions, notation and examples	. 105
4.4.2 The learning algorithm	. 115
4.5 Soundness and complexity	. 120
4.5.1 Soundness	. 121
4.5.2 Complexity	. 121
4.6 Application to program verification	. 122
4.6.1 Applying the learning algorithm	. 122
4.6.2 Derivation of loop invariants	. 123
4.7 Comparison with other methods	. 128
4.7.1 Representation language and learning methodology	. 128
4.7.2 Performance comparison	. 131
V. MECHANIZING MATHEMATICAL INDUCTION	. 136
5.1 Introduction	. 136
5.2 Related work	. 137
5.3 Description of the first method	. 140
5.3.1 Discovering a well-founded ordering	. 141
5.3.2 Using the induction principle	. 142
5.3.3 Finding one induction hypothesis	. 143
5.3.4 Finding more than one induction hypothesis	. 145
5.4 Description of the second method	. 145
5.5 Comparison with other methods	. 149
VI. CONCLUSION	. 151
6.1 Summary	. 151
6.2 Extensions	. 152
6.2.1 Automatic generation of loop invariants	. 152
6.2.2 Learning from examples	. 153
6.2.3 Mechanizing mathematical induction	. 154
Appendix	. 155
References	. 179

LIST OF FIGURES

Figure 3.1:	Calculating the quotient and remainder of two numbers	52
Figure 3.2:	Calculating the g.c.d. of two numbers	77
Figure 3.3:	Multiplying two numbers by repeated addition	84
Figure 4.1:	Blocks for Example 4.1	106
Figure 4.2:	Blocks for Example 4.2	107
Figure 4.3:	Clause graph for Example 4.2	109
Figure 4.4:	Argument graph for Example 4.2	110
Figure 4.5:	Subgraph of clause graph for Example 4.2	112
Figure 4.6:	Clause graph for Example 4.3	113
Figure 4.7:	Argument graph for Example 4.3	114
Figure 4.8:	Flowchart program for Example 4.5	124
Figure 4.9:	Flowchart program for Example 4.6	126
Figure 4.10:	Three blocks examples	129
Figure A.1:	Clause graph for E_1 and E_2	160
Figure A.2:	Argument graph for E_1 and E_2	161
Figure A.3:	Subgraph of clause graph for E_1 and E_2	162
Figure A.4:	Clause graph for E_3 and EX	163
Figure A.5:	Argument graph for E_3 and EX	164
Figure A.6:	Subgraph of clause graph for E_3 and EX	165

1. Introduction

This dissertation is concerned with the development of a novel unskolemization technique and its application to three different fields, namely program verification, machine learning, and mathematical induction. Unskolemization can be loosely defined as the process of replacing terms consisting of function constants by existentially quantified variables. An unskolemization algorithm is developed for the purpose of deriving logical consequences of first-order formulas. It is then shown how such an algorithm can be put to use in these three areas.

One of the foremost problems in automating program verification is the need for deriving loop invariants for loops in programs. None of the existing program verifiers can automatically generate loop invariants for program loops, although some of them do provide limited support in deriving these invariants. Thus automatic program verification shifts the sometimes onerous task of finding loop invariants to the user. We describe an algorithm based on the unskolemization algorithm mentioned above for automatically generating loop invariants. This algorithm is complete in the sense that if a loop invariant exists for a particular program loop in a given first-order language relative to a given finite set of first-order axioms, then the algorithm can find it. Of course, not all theories of interest can be expressed by a finite collection of first-order axioms.

At present, to make a computer perform a task, one has to write a complete and correct algorithm for that task, and program the algorithm into the computer. These activities involve a tedious and time-consuming effort by specially trained personnel. Current computer systems cannot improve significantly on the basis of past mistakes, nor can they acquire new abilities by observing and imitating experts. Machine learning research strives to open the possibility of instructing computers in such new ways. We have developed an algorithm, based on our unskolemization algorithm, for learning from examples expressed in first-order logic. The algorithm can be used to make the derivation of loop invariants more efficient, as well as in other traditional fields like the blocks world, where it can produce common descriptions of several situations.

The principle of mathematical induction cannot be expressed in first-order

logic; it belongs to the realm of second-order logic, as it involves quantification over predicates. However, unlike first-order logic, no complete proof systems exist for second-order logic. To handle mathematical induction within the framework of first-order logic, it is necessary to find an induction schema for each theorem to be proved by induction. The scenario becomes more complicated when a theorem to be proved by induction in turn depends on some other theorem which is to be proved by induction. We examine these problems and describe solutions, including one using our unskolemization algorithm.

In the next section, a brief introduction to the subject of first-order logic is given to equip readers unfamiliar with the subject with sufficient knowledge to read this dissertation. An outline of the organization of this dissertation is given at the end of this chapter.

1.1 A brief introduction to first-order logic

1.1.1 Preliminaries

The purpose of this section is to familiarize the reader with some of the concepts of first-order logic which are necessary for understanding the remainder of the material in this document. Many of these definitions are taken from [Manna 74] and [Chang and Lee 73].

First-order logic is a formal language whose purpose is to symbolize logical arguments in mathematics. The sentences in this language are called *well-formed formulas (wffs)*. By giving a meaning to, or “interpreting”, the symbols in a wff we obtain a *statement* which is either true or false. We can associate many different interpretations with the same wff and therefore obtain a class of statements where each statement is either true or false. We are interested mainly in two very restricted subclasses of the wffs, those that yield a true statement for every possible interpretation, and those that yield a false statement for every possible interpretation.

There are some symbols in first-order logic which have fixed meaning :

1. $(\exists x)A$ stands for “there exists some x such that A is true”.
2. $(\forall x)A$ stands for “for every element x , A is true”.
3. $P \wedge Q$ stands for “ P and Q are true”.
4. $P \vee Q$ stands for “ P or Q is true”.
5. $\neg P$ stands for “ P is not true”.
6. $P \rightarrow Q$ stands for “if P is true, then Q is true”; here P is called the *antecedent* and Q the *consequent*.

For example, the wff

$$(\forall x)((P(x) \vee Q(x)) \wedge (\forall y)(\exists z)(G(z, y)))$$

has the following meaning : for every x , either $P(x)$ or $Q(x)$ (or both) are true, and for every y , there exists z such that $G(z, y)$ is true.

An interpretation of this wff is given by specifying a non-empty set D and then assigning a unary predicate (mapping D into $\{true, false\}$) to P , a unary predicate to Q , and a binary predicate (mapping $D \times D$ into $\{true, false\}$) to G . For example, if we choose D to be the set N of all natural numbers and we let $P(x)$ be the predicate “ x is even”, $Q(x)$ be the predicate “ x is odd”, and $G(x, y)$ be the predicate “ $x > y$ ”, then the above wff becomes :

For all x , x is either even or odd, and for every y , there exists z such that $z > y$. This statement is easily seen to be true, since all natural numbers are either odd or even, and the set of natural numbers has no upper bound in the natural numbers.

We say that x, y , and z are *quantified* in the above wff, since they are preceded by the quantifiers “ \exists ” or “ \forall ”.

We now formally define the syntax and semantics of first-order logic.

1.1.2 Syntax

The five different kinds of symbols from which our sentences are constructed are listed below.

1. **Truth symbols** : T and F (*true* and *false*)
2. **Connectives** : \wedge (and), \vee (or), \equiv (equivalence), \neg (not), \rightarrow (implication).
3. **Quantifiers** : \forall (universal quantifier) and \exists (existential quantifier)
4. **Constants** :
 - (a) n -ary predicate constants P_i^n ($i \geq 1, n \geq 0$); P_i^0 is called a propositional constant
 - (b) n -ary function constants f_i^n ($i \geq 1, n \geq 0$); f_i^0 is called an individual constant and is also denoted by a_i
5. **Variables** : individual variables x_i

Using these symbols we recursively define three classes of expressions : *terms*, *atoms*, and *well-formed formulas (wffs)*.

1. **Terms** :

- (a) Each individual constant a_i and each individual variable x_i is a term.

(b) If t_1, t_2, \dots, t_n ($n \geq 1$) are terms, then so is $f_i^n(t_1, t_2, \dots, t_n)$.

2. Atoms :

(a) T and F are atoms.

(b) Each propositional constant P_i^0 is an atom.

(c) If t_1, t_2, \dots, t_n ($n \geq 1$) are terms, then $P_i^n(t_1, t_2, \dots, t_n)$ is an atom.

3. Well-formed formulas (wffs) :

(a) Each atom is a wff.

(b) If A, B and C are wffs, then so are $(\neg A)$, $(A \rightarrow B)$, $(A \wedge B)$, $(A \vee B)$, and $(A \equiv B)$.

(c) If x_i is a variable and A is a wff, then $((\forall x_i)A)$ and $((\exists x_i)A)$ are wffs.

In what follows, several straightforward abbreviations are used. Since the superscripts in f_i^n, P_i^n are used only to indicate the number of arguments, they are always omitted. The subscripts are also omitted whenever their omission can cause no confusion. For simplicity we usually use additional symbols : a, b, c, \dots for individual constants; f, g, h, \dots for function constants; capital letters for predicate constants (also called predicates); and u, v, w, x, \dots for individual variables. Also we usually omit parentheses whenever their omission can cause no confusion; in particular, we usually write $(\exists x)$ and $(\forall x)$ as $\exists x$ and $\forall x$ without parentheses. Sometimes we use brackets [] or braces { } rather than parentheses () for clarity. We assume that $(\exists x)$, $(\forall x)$, and \neg bind tighter than any other connective, i.e. they are always applied to the smallest possible scope (atom or parenthesized expression).

For a wff of the form $(\forall x)A$, we say that the occurrence of the variable x in $(\forall x)$ is *universally quantified*, A is the *scope of $(\forall x)$* , and every occurrence of x in A is *bound by $(\forall x)$* . Similarly, for $(\exists x)A$, we say that the occurrence of the variable x in $(\exists x)$ is *existentially quantified*, A is the *scope of $(\exists x)$* , and every occurrence of x in A is *bound by $(\exists x)$* . Every occurrence of a variable in a wff which is not quantified or bound is said to be a *free occurrence*. A variable x is said to be a *free variable* of a wff if there are free occurrences of x in the wff. A wff with no free variables is said to be *closed*.

The class of wffs described here are the wffs of first-order logic. One subclass of these wffs, consisting of the *propositional calculus* formulas is of special interest and is obtained by restricting the set of constant symbols to be propositional constants and the set of variables to be empty. A class of wffs containing first-order logic wffs, consisting of *second-order logic* formulas, is obtained by allowing function and predicate variables in addition to individual variables in wffs.

1.1.3 Semantics

We can assign a meaning to each wff by “interpreting” the constant symbols and free variables in it. By associating different interpretations with a given wff, we obtain different *statements*, where each statement is either *true* or *false*. In this section we shall discuss the notion of an interpretation of a wff and the statement generated by it.

Let D be any nonempty set; then D^n ($n \geq 1$) represents the set of all ordered n -tuples of elements of D . An n -ary function over D ($n \geq 1$) is a total function mapping D^n into D . An n -ary predicate over D ($n \geq 1$) is a total function mapping D^n into $\{true, false\}$. In the case where $n = 0$, a 0-ary function over D denotes a fixed element of D , while a 0-ary predicate over D denotes a fixed truth value (*true* or *false*).

An **interpretation** \mathcal{I} of a wff A is a triple $(D, \mathcal{I}_c, \mathcal{I}_v)$ where

1. D is a nonempty set which is called the domain of the interpretation.
2. \mathcal{I}_c indicates the assignments to the constants of A :
 - (a) We assign an n -ary function over D to each function constant f_i^n ($n \geq 0$) which occurs in A . In particular (case $n = 0$), each individual constant a_i is assigned some element of D .
 - (b) We assign an n -ary predicate over D to each predicate constant P_i^n ($n \geq 0$) which occurs in A . In particular (case $n = 0$), each propositional constant is assigned the truth value *true* or *false*.
3. \mathcal{I}_v indicates the assignments to the free variables of A : each free variable x in A is assigned some element of D .

For a given interpretation \mathcal{I} of a wff A , the pair (A, \mathcal{I}) indicates a statement (sometimes called an interpreted wff) which has a truth value *true* or *false*. We obtain this truth value by first applying the assignments of \mathcal{I}_c to all the constant symbols in A and the assignments of \mathcal{I}_v to all free occurrences of variable symbols in A , and then using the meaning (semantics) of the truth symbols, connectives, operators, and quantifiers as explained below.

1. The meaning of the truth symbols :

The meaning of T is *true* and of F is *false*.

2. The meaning of connectives :

(a) The connective \neg (not) stands for a unary function mapping $\{true, false\}$ into $\{true, false\}$ as follows :

$\neg true$ is *false*

$\neg false$ is *true*.

(b) The connectives \rightarrow (implication), \wedge (and), \vee (or), and \equiv (equivalence) stand

for binary functions mapping $\{true, false\} \times \{true, false\}$ into $\{true, false\}$ as follows :

$true \rightarrow false$ is *false*

$true \rightarrow true$, $false \rightarrow true$, and $false \rightarrow false$ are *true*

$true \wedge true$ is *true*

$true \wedge false$, $false \wedge true$, and $false \wedge false$ are *false*

$false \vee false$ is *false*

$true \vee true$, $true \vee false$, and $false \vee true$ are *true*

$true \equiv true$ and $false \equiv false$ are *true*

$true \equiv false$ and $false \equiv true$ are *false*

3. The meaning of the quantifiers :

We consider wffs of the form $(\forall x)A$ and $(\exists x)A$. Since such a wff might have some free occurrences of variables, we have to consider its value for some fixed assignment of values to those free occurrences.

(a) The quantifier \forall (universal quantifier) in the wff $(\forall x)A$ stands for the words “for all x , A is true”. The value of $(\forall x)A$ is *true*, if for all elements a of D , the value of A (with a assigned to all occurrences of x) is *true*; otherwise, the value of $(\forall x)A$ is *false*.

(b) The quantifier \exists (existential quantifier) in the wff $(\exists x)A$ stands for the words “there exists x such that A is true”. The value of $(\exists x)A$ is *true* if there exists an element a of D such that the value of A (with a assigned to all occurrences of x) is *true*; otherwise, the value of $(\exists x)A$ is *false*.

A wff may yield the value *true* for some interpretations and the value *false* for some other interpretations. We are interested mainly in two types of wffs : those that yield the value *true* for every possible interpretation, called *valid* wffs, and those that yield the value *false* for every possible interpretation, called *unsatisfiable* wffs. In other words :

Definition. A wff A is said to be *valid* if it yields the value *true* for every interpretation. Thus, a wff is *nonvalid* if and only if there exists some interpretation for which A yields the value *false*; such an interpretation is called a *countermodel* for A . A wff A is said to be *unsatisfiable* if it yields the value *false* for every interpretation. A wff is *satisfiable* if and only if there exists some interpretation for which A yields the value *true*; such an interpretation is called a *model* for A .

There is one important relation between the two notions : a wff A is *valid* if and only if $\neg A$ is *unsatisfiable*.

Example 1.1 The wff $P \vee \neg P$ is valid. This is because, in an interpretation of this wff, the propositional constant P must be assigned either the value *true* or the

value *false*. If P is assigned the value *true*, then since P is *true*, $\neg P$ is *false*, and therefore by our definition of \vee , $P \vee \neg P$ is *true*. On the other hand, if P is assigned the value *false*, then $\neg P$ is *true* and therefore $P \vee \neg P$ is again *true*. Thus the wff is *true* in any interpretation and therefore it is valid. •

Example 1.2 The wff $P \wedge \neg P$ is unsatisfiable. This is because, in an interpretation of this wff, the propositional constant P must be assigned either the value *true* or the value *false*. If P is assigned the value *true*, then since P is *true*, $\neg P$ is *false*, and therefore by our definition of \wedge , $P \wedge \neg P$ is *false*. On the other hand, if P is assigned the value *false*, then $\neg P$ is *true* and therefore $P \wedge \neg P$ is again *false*. Thus the wff is *false* in any interpretation and therefore it is unsatisfiable. •

Definition. Given two wffs A and B , we say $A \models B$ (read “ A satisfies B ”) if and only if every model for A is a model for B . Clearly, $A \models B$ if and only if $A \rightarrow B$ is valid.

1.1.4 Normal forms and related definitions

In this section, we describe some normal forms of wffs. These forms are called *normal* because every wff can be transformed into an equivalent wff having any one of these forms. The reason for these transformations is to simplify proof procedures, which will be discussed later.

Definition. A *literal* is an atom or the negation of an atom. The sign of a literal L is said to be positive if L is an atom and negative if L is the negation of an atom.

Definition. A *conjunction* of literals is a wff of the form $L_1 \wedge L_2 \wedge \dots \wedge L_n$, where each L_i , $1 \leq i \leq n$, is a literal.

Definition. A *disjunction* of literals is a wff of the form $L_1 \vee L_2 \vee \dots \vee L_n$, where each L_i , $1 \leq i \leq n$, is a literal.

Definition. A wff F in the first-order logic is said to be in *prenex normal form* if and only if the formula F is in the form

$$(Q_1 x_1) \dots (Q_n x_n) (M)$$

where every $(Q_i x_i)$, $1 \leq i \leq n$ is either $(\forall x_i)$ or $(\exists x_i)$, and M is a wff containing no quantifiers. $(Q_1 x_1) \dots (Q_n x_n)$ is called the *prefix* and M is called the *matrix* of the formula F .

Some examples of wffs in prenex normal form are :

$$(\forall x)(\forall y)(P(x, y) \wedge Q(y)),$$

$$(\forall x)(\forall y)(\neg P(x, y) \rightarrow Q(y)),$$

$$(\forall x)(\forall y)(\exists z)(Q(x, y) \rightarrow R(z)).$$

We present the following theorem without proof :

Theorem 1.1 Every wff can be transformed into an equivalent wff in prenex normal form. (For proof see [Chang and Lee 73].)

Definition. A wff F in first-order logic is said to be in *prenex conjunctive normal form* if and only if F is in prenex normal form, and the matrix M of F has the form $F_1 \wedge F_2 \wedge \dots \wedge F_n$, $n \geq 1$, where each of F_1, F_2, \dots, F_n is a disjunction of literals.

Definition. A wff F in first-order logic is said to be in *prenex disjunctive normal form* if and only if F is in prenex normal form, and the matrix M of F has the form $F_1 \vee F_2 \vee \dots \vee F_n$, $n \geq 1$, where each of F_1, F_2, \dots, F_n is a conjunction of literals.

Every wff can be transformed into an equivalent wff in prenex conjunctive or prenex disjunctive normal form. Procedures for transforming wffs into prenex conjunctive and disjunctive normal forms can be found in [Chang and Lee 73].

Skolem standard form

We now describe how to transform a wff into a standard form known as the *Skolem standard form*. Briefly, the objective of this transformation is to eliminate the existential quantifiers and existentially quantified variables from a wff, thus making the wff more readily amenable to mechanical manipulation. Existentially quantified variables are replaced by new functions (called Skolem functions) in such a way that the unsatisfiability of the formula is preserved; in other words, if a wff F is unsatisfiable, then the Skolem form of F is also unsatisfiable and vice-versa. The process does not preserve validity, however.

The procedure for transforming a wff into Skolem standard form is given below. Suppose a wff F is given.

1. Transform F into prenex conjunctive normal form.
2. Eliminate existential quantifiers from the prefix of F as follows : suppose the prefix of F is $(Q_1 x_1) (Q_2 x_2) \dots (Q_n x_n)$. Suppose Q_r is an existential quantifier in this prefix, $1 \leq r \leq n$. If no universal quantifier appears before Q_r , choose a new constant c different from other constants occurring in M (where M is the matrix of F), replace all x_r appearing in M by c , and delete $(Q_r x_r)$ from the prefix. If $Q_{r_1}, Q_{r_2}, \dots, Q_{r_m}$ are all the universal quantifiers appearing before Q_r , $1 \leq r_1 < r_2 < \dots < r_m < r$, choose a new m -ary function symbol f different from other function symbols, replace all x_r in M by $f(x_{r_1}, x_{r_2}, \dots, x_{r_m})$, and delete $(Q_r x_r)$ from the prefix. The intuitive reason for performing this step is that since x_r is an existentially quantified variable whose value depends on $x_{r_1}, x_{r_2}, \dots, x_{r_m}$, it can be regarded as a function of the variables $x_{r_1}, x_{r_2}, \dots, x_{r_m}$. After the above process is applied to all the existential quantifiers in the prefix, the last formula obtained is a *Skolem standard form* of the wff F . The constants and functions used to replace the

existential variables are called *Skolem functions* or *Skolem symbols*, and the process of transforming a wff into Skolem standard form is called *Skolemization*. If we are given a wff F , then its Skolem standard form is denoted by $Sk(F)$.

Example 1.3 We demonstrate how a formula in prenex conjunctive normal form is transformed into Skolem standard form. Let

$$F = (\exists x)(\forall y)(\exists z)((\neg P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z))).$$

Here $(\exists x)$ is not preceded by any universal quantifier in the prefix of F , therefore we replace x everywhere by a new constant c ; and $(\exists z)$ is preceded by $(\forall y)$, therefore we replace z everywhere by a new unary function $f(y)$, obtaining the Skolem standard form :

$$Sk(F) = (\forall y)((\neg P(c, y) \vee R(c, y, f(y))) \wedge (Q(c, f(y)) \vee R(c, y, f(y)))) . \bullet$$

Definition. A *clause* is a disjunction of zero or more literals.

Henceforth, we shall regard a set of literals as synonymous with a clause. For example, $P \vee Q \vee \neg R = \{P, Q, \neg R\}$. This notation is consistent with set-theoretical notation, due to the fact that disjunction is idempotent (i.e. $P \vee P \equiv P$), commutative, and associative. A clause containing one literal is called a unit clause; when a clause contains no literals, it is called the empty clause. Since the empty clause has no literal that can be satisfied by an interpretation, the empty clause is always false. A set of clauses S is regarded as the conjunction of all clauses in S , where every variable in S is regarded as universally quantified. The formula " $\forall F$ ", where F is a wff without quantifiers, denotes the wff F with all variables in F universally quantified. Under this interpretation, a set of clauses is unsatisfiable, valid, or satisfiable according as the wff obtained by forming the conjunction of all the clauses in the set is unsatisfiable, valid, or satisfiable. By the above convention, the Skolem standard form of a formula can be represented by a set of clauses. For example, the Skolem standard form obtained in the last example can be written in clause form as the set

$$\{\{\neg P(c, y), R(c, y, f(y))\}, \{Q(c, f(y)), R(c, y, f(y))\}\}.$$

The motivation for transforming a formula into Skolem standard form is made clear by the following theorem :

Theorem 1.2 Let S be a set of clauses that represents a Skolem standard form of a wff F . Then F is unsatisfiable if and only if S is unsatisfiable. (For proof see [Chang and Lee 73].)

In the next section we shall see how the Skolem standard form of a wff is used in proof procedures. Before doing so, we need a few more definitions.

Definition. A term is called a *ground term* if it contains no variables.

Definition. A *substitution* is a finite set of the form $\{t_1/v_1, \dots, t_n/v_n\}$, where

every v_i is a variable, every t_i is a term different from v_i , and all the v_i s are distinct. When t_1, \dots, t_n are ground terms, the substitution is called a *ground substitution*. The substitution that consists of no elements is called the *empty substitution* and is denoted by ϵ . We shall use lower case Greek letters to represent substitutions.

Example 1.4 The following sets are substitutions :

$$\{f(a)/x, z/y, w/z\}, \quad \{f(z)/x, g(z)/y, g(a)/z\}. \bullet$$

Definition. Let $\theta = \{t_1/v_1, \dots, t_n/v_n\}$ be a substitution and E be an expression. Then $E\theta$ is an expression obtained from E by simultaneously replacing each occurrence of the variable v_i , $1 \leq i \leq n$, in E by the term t_i . $E\theta$ is called an *instance* of E .

Example 1.5 Let $\theta = \{g(a)/x, b/y, f(c)/z\}$ and $E = P(x, y, z)$. Then $E\theta = P(g(a), b, f(c))$. •

Definition. Let $\theta = \{t_1/x_1, \dots, t_n/x_n\}$ and $\sigma = \{s_1/y_1, \dots, s_m/y_m\}$ be two substitutions. Then the *composition* of θ and σ is the substitution, denoted by $\theta \circ \sigma$, that is obtained from the set

$$\{t_1\sigma/x_1, \dots, t_n\sigma/x_n, s_1/y_1, \dots, s_m/y_m\}$$

by deleting any element $t_j\sigma/x_j$ for which $t_j\sigma = x_j$, and any element s_i/y_i such that y_i is among $\{x_1, x_2, \dots, x_n\}$.

Example 1.6 Let $\theta = \{w/x, f(u)/y, f(c)/z\}$ and $\sigma = \{x/w, b/u, a/z\}$.

Then $w\sigma = x$, $f(u)\sigma = f(b)$, $f(c)\sigma = f(c)$.

Therefore $\theta \circ \sigma = \{f(b)/y, f(c)/z, x/w, b/u\}$.

Note that the elements $w\sigma/x$ (i.e. x/x) and a/z were deleted from the above set, according to the definition of composition given above.

Definition. Given two wffs A and B , A and B are *variants* of each other if and only if there exist substitutions θ, σ such that $A\theta = B$ and $A = B\sigma$.

Clearly, if wffs A and B are variants, then they are equivalent to each other.

In what follows, we shall often have to unify, or match, two or more expressions. That is, we have to find a substitution that can make several expressions identical. Therefore, we now consider the unification of expressions.

Definition. A substitution σ is called a *unifier* for a set of expressions $\{E_1, E_2, \dots, E_k\}$ if and only if $E_1\sigma = E_2\sigma = \dots = E_k\sigma$. The set $\{E_1, E_2, \dots, E_k\}$ is said to be *unifiable* if there is a unifier for it.

Example 1.7 The set $\{\neg Q(f(a), g(y)), \neg Q(x, g(b))\}$ is unifiable since the substitution $\theta = \{f(a)/x, b/y\}$ is a unifier for the set. •

Definition. A unifier σ for a set of expressions $\{E_1, E_2, \dots, E_k\}$ is a *most general unifier* if and only if for each unifier θ for the set there is a substitution λ such that $\theta = \sigma \circ \lambda$.

A large number of algorithms for finding the most general unifier of a set of expressions have been developed in the past. We will not present any unification algorithms here but instead refer the interested reader to the literature. Any introductory theorem-proving text such as [Chang and Lee 73] or [Loveland 78] contains some elementary unification algorithms.

1.1.5 Proof procedures

In this section we shall discuss proof procedures for wffs of first-order logic. Leibniz (1646–1716) was the first to search for a general decision procedure for verifying the validity or unsatisfiability of a formula. Peano and Hilbert continued this search, and finally Church and Turing were able to prove independently in 1936 that there is no general decision procedure to verify the validity of formulas of the first-order logic. However, there are proof procedures which can verify that a formula is valid if it is indeed valid. For invalid formulas, these procedures cannot be guaranteed to terminate. Given the result of Church and Turing, this is the best that we can hope to achieve.

A very important approach to mechanical theorem proving was given by Herbrand in 1930. Recall that by definition, a valid wff is a wff that is true under all interpretations. Herbrand developed an algorithm to find an interpretation that can falsify a given wff. However, if the given wff is indeed valid, no such interpretation can exist and his algorithm will halt after a finite number of trials. Herbrand's theorem is the basis for many modern theorem-proving procedures, including resolution. A complete discussion and proof of Herbrand's theorem is beyond the scope of this work; we will only state the theorem and refer the reader to [Chang and Lee 73] for a proof.

Definition. Let S be a set of clauses, and let H_0 be the set of function constants appearing in S . If no constant appears in S , then H_0 is to consist of a single constant, say $H_0 = \{a\}$. For $i = 0, 1, 2, \dots$, let H_{i+1} be the union of H_i and the set of all terms of the form $f^n(t_1, t_2, \dots, t_n)$ for all n -ary functions f^n occurring in S , where t_j , $1 \leq j \leq n$, are members of the set H_i . Then each H_i is called the i -level constant set of S , and H_∞ , or $\lim_{i \rightarrow \infty} H_i$, is called the *Herbrand universe* of S .

Definition. A *ground instance* of a clause C of a set S of clauses is a clause obtained by replacing all variables in C by members of the Herbrand universe of S . A clause which does not contain any variables is called a *ground clause*.

Herbrand's Theorem. A set S of clauses is unsatisfiable if and only if there is a finite unsatisfiable subset S' of ground instances of clauses of S . (For proof see [Chang and Lee 73].)

We will now describe the resolution principle for proving the unsatisfiability of a set of clauses. The method was discovered in 1965 by Robinson [Robinson 65] and is a landmark in the history of theorem-proving. The resolution principle is an inference rule that generates resolvents from a set of clauses (defined below). It is more efficient than earlier proof procedures such as the Davis and Putnam procedure. The resolution principle is really a refutation procedure, in that it is used for proving that a wff is unsatisfiable, rather than proving that a wff is valid. However, as stated earlier, a wff F is valid if and only if $\neg F$ is unsatisfiable. Thus a refutation procedure can be used as a validity-proving procedure.

Suppose therefore that we are given a wff F which must be shown to be valid. We first negate F ; we must now show that $\neg F$ is unsatisfiable. $\neg F$ is then converted into Skolem standard form and written as a set of clauses S . Recall that this conversion preserves unsatisfiability, i.e. $\neg F$ is unsatisfiable if and only if its Skolem standard form S is unsatisfiable. S is then shown to be unsatisfiable using the resolution procedure, which is described below.

Definition. If two or more literals of a clause C have a most general unifier σ , then $C\sigma$ is called a *factor* of C .

Definition. Let C_1 and C_2 be two clauses (called *parent clauses*) with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2 respectively. If L_1 and $\neg L_2$ have a most general unifier σ , then the clause

$$(C_1\sigma - \{L_1\}\sigma) \cup (C_2\sigma - \{L_2\}\sigma)$$

is called a *binary resolvent* of C_1 and C_2 . The literals L_1 and L_2 are called the *literals resolved upon*.

Example 1.8 Let $C_1 = \{P(f(x)), Q(x)\}$, $C_2 = \{\neg P(f(a)), R(y)\}$. Choose $L_1 = P(f(x))$ and $L_2 = \neg P(f(a))$. Since $\neg L_2 = P(f(a))$, L_1 and L_2 have the most general unifier $\sigma = \{a/x\}$ (this can easily be seen to be the only unifier of L_1 and L_2). Therefore,

$$\begin{aligned} & (C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma) \\ &= (\{P(f(a)), Q(a)\} - \{P(f(a))\}) \cup (\{\neg P(f(a)), R(y)\} - \{\neg P(f(a))\}) \\ &= \{Q(a)\} \cup \{R(y)\} \\ &= \{Q(a), R(y)\}. \end{aligned}$$

Thus $\{Q(a), R(y)\}$ is a binary resolvent of C_1 and C_2 . $P(f(x))$ and $\neg P(f(a))$ are the literals resolved upon. •

Definition. A *resolvent* of (parent) clauses C_1 and C_2 is one of the following binary resolvents :

1. a binary resolvent of C_1 and C_2 ,
2. a binary resolvent of C_1 and a factor of C_2 ,

3. a binary resolvent of a factor of C_1 and C_2 ,
4. a binary resolvent of a factor of C_1 and a factor of C_2 .

Resolution is used for proving that a given set S of clauses is unsatisfiable. The resolution procedure consists of generating resolvents of a set S of clauses, then generating resolvents of S and these resolvents, and so on, until the empty clause is generated. A clause C is said to be generated by resolution from S if and only if it can be generated by a sequence of resolutions from S ; in this case, we write $S \vdash C$ (read " S derives C "). The set of all clauses which can be generated by resolution from S is denoted by $Res(S)$.

Definition. A wff B is said to be a *logical consequence* of a wff A if and only if $A \rightarrow B$ is valid; or in other words, B is a *logical consequence* of a wff A if and only if every model for A is a model for B .

Theorem 1.3 A resolvent R of two clauses C_1 and C_2 is a logical consequence of C_1 and C_2 . (For proof see [Chang and Lee 73].) In other words, $(C_1 \wedge C_2 \vdash R) \rightarrow (C_1 \wedge C_2 \models R)$.

This theorem establishes the soundness of the resolution principle. To see this, note that if the empty clause can be derived from a set S of clauses by resolution, then by the previous theorem the empty clause is a logical consequence of S . However, the empty clause represents the value *false* (since it cannot be satisfied by any interpretation). This means that $S \rightarrow \text{false}$ is valid for all interpretations of S , which in turn means that S is unsatisfiable under all interpretations.

Apart from the soundness of resolution, we also have the following result :

Theorem 1.4 Resolution is *complete*; in other words, a set S of clauses is unsatisfiable only if there is a deduction of the empty clause from S . (For proof see [Chang and Lee 73].) In other words, $(S \models \{ \}) \rightarrow (S \vdash \{ \})$.

This establishes that resolution is a sound and complete refutation procedure, or in other words, if S is a set of clauses, then $(S \models \{ \}) \equiv (S \vdash \{ \})$.

Example 1.9 Let $S = \{ \{P, Q\}, \{ \neg P, Q\}, \{P, \neg Q\}, \{ \neg P, \neg Q\} \}$. We show that S is unsatisfiable by generating the empty clause from S by resolution as follows :

- | | |
|------------------|---|
| 1. $\{Q\}$ | from clauses $\{P, Q\}, \{ \neg P, Q\}$ |
| 2. $\{ \neg Q\}$ | from clauses $\{P, \neg Q\}, \{ \neg P, \neg Q\}$ |
| 3. $\{ \}$ | from clauses 1 and 2. |

Clause 3 is the empty clause, which establishes that the set S is unsatisfiable. •

Thus we have seen that the empty clause can be generated from an unsatisfiable set of clauses by resolution. We now introduce *paramodulation*, which is essentially an inference rule for equality. In the case where a set of clauses includes the equality

relation under the usual interpretation of equality, paramodulation and resolution will always generate the empty clause from a set of clauses which are unsatisfiable under the usual equality axioms. The formal definition of paramodulation and the equality axioms are given below.

Equality axioms : Let S be a set of clauses. Then the set of equality axioms for S contains the following clauses :

1. $\forall x(x = x)$
2. $\forall x\forall y(x \neq y \vee y = x)$
3. $\forall x\forall y\forall z(x \neq y \vee y \neq z \vee x = z)$
4. $x_j \neq x_0 \vee \neg P(x_1, \dots, x_j, \dots, x_n) \vee P(x_1, \dots, x_0, \dots, x_n)$ for $j = 1, \dots, n$, for every n -ary predicate symbol P occurring in the given set of clauses S
5. $x_j \neq x_0 \vee \neg f(x_1, \dots, x_j, \dots, x_n) \vee f(x_1, \dots, x_0, \dots, x_n)$ for $j = 1, \dots, n$, for every n -ary function symbol f occurring in the given set of clauses S .

Definition. Let C_1 and C_2 be two clauses (called *parent clauses*) with no variables in common. If C_1 is $L[t] \vee C'_1$, and C_2 is $r = s \vee C'_2$, where $L[t]$ is a literal containing the term t and C'_1 and C'_2 are clauses, and if t and r have a most general unifier σ , then the clause

$$L\sigma[s\sigma] \cup C'_1\sigma \cup C'_2\sigma,$$

where $L\sigma[s\sigma]$ denotes the result obtained by replacing one single occurrence of $t\sigma$ in $L\sigma$ by $s\sigma$, is called a *binary paramodulant* of C_1 and C_2 .

Definition. A *paramodulant* of clauses C_1 and C_2 is one of the following binary paramodulants :

1. a binary paramodulant of C_1 and C_2 ,
2. a binary paramodulant of C_1 and a factor of C_2 ,
3. a binary paramodulant of a factor of C_1 and C_2 ,
4. a binary paramodulant of a factor of C_1 and a factor of C_2 .

1.1.6 Theories

We have so far been discussing wffs of first-order logic and methods of determining their validity. We are often interested in wffs which are valid in some specific theory, such as number theory or group theory. We make these concepts precise below :

Definition. A non-empty set T of wffs of first-order logic is called a *theory* if and only if

- (i) There exists at least one model for T

(ii) All logical consequences of T are already in T .

The above definition provides a convenient way of creating a theory : take a set of wffs with a model and form the closure under logical consequence.

Theorem 1.5 For every interpretation I , the set $Th(I)$ of all wffs valid in I is a theory.

Proof : We need to show that conditions (i) and (ii) of the above definition are satisfied by $Th(I)$. Condition (i) is trivially satisfied, since the interpretation I is a model for $Th(I)$. Now consider any logical consequence L of $Th(I)$. Every model of $Th(I)$ is a model for L , by definition; therefore in particular, I is a model for L . Hence L is valid in I , and therefore L belongs to $Th(I)$, by definition of $Th(I)$; this proves condition (ii). •

Definition. A theory T is said to be *axiomatizable* if there exists a decidable set $D \subseteq T$ such that T is exactly the set of all wffs implied by D . The wffs of D are called the *axioms* of the theory T .

Note. Henceforth we will use the term “formula” to mean “well-formed formula”, unless otherwise indicated.

1.2 Outline of this dissertation

In the next chapter, we show how logical consequences of first-order formulas can be derived by resolution and unskolemization. An unskolemization algorithm for replacing Skolem and sometimes non-Skolem functions by existentially quantified variables is presented and its properties analyzed. A number of examples are given to illustrate the working of the algorithm.

Chapter 3 introduces the topic of program verification and shows how the unskolemization algorithm of Chapter 2 can be used in conjunction with an iterative algorithm to mechanically generate loop invariants for the purpose of verifying the partial correctness of a program. A detailed synopsis of past work in the area of program verification is included in this chapter.

In Chapter 4, we survey some of the past work in the field of machine learning, and show how the theory developed in Chapter 2 can be applied to the subject of machine learning from positive examples in first-order logic. A learning algorithm is given which makes use of the unskolemization algorithm of Chapter 2. We demonstrate how the learning algorithm can be used to make the generation of loop invariants in Chapter 3 more efficient. The algorithm is analyzed and compared with other work in the same field.

Finally, in Chapter 5, we demonstrate the application of our method for generating logical consequences of first-order formulas to the mechanization of mathematical induction. We examine different ways of automatically generating the inductive hypotheses for proving theorems by induction. We describe two complete methods for generating the inductive hypotheses for certain types of theorems. The first is based on our resolution and unskolemization method. The second method is based on the fact that the proof of a theorem which requires inductive hypotheses for its proof can be deduced by examining proofs of different ground instances of the theorem.

Chapter 6 concludes this dissertation and discusses the relevance of this research, as well as directions for future work.

2. Finding logical consequences using unskolemization

2.1 Objective and motivation

In this chapter, we will develop a method for finding logical consequences of first-order formulas. The method is based on resolution and on an unskolemization algorithm. Suppose we are given a first-order formula H , and we want to find a certain consequence W of H , which is unknown. It may not be possible to derive W from H by resolution, without using tautologies and unskolemization, as will be shown in Section 2.2.1. Since the use of tautologies is undesirable (due to the enormous increase in search space that it creates), we will not attempt to derive W from H , but instead will try to derive a formula F with the property that

$$H \rightarrow F \rightarrow W.$$

However, if this is the only constraint on F , then why not take $F = H$? One obvious reason is that H may be infinite. Also, we want F to be as “close” as possible to W , in a certain sense. To define the concept of “closeness”, we will define a relation “more general than” on first-order formulas and will derive a formula F from H such that $H \rightarrow F \rightarrow W$ and such that F is “more general than” W . The relation “more general than” is defined in such a way that the number of first-order formulas F which satisfy a given syntactic condition and are more general than a given first-order formula W is finite up to variants. Thus we can only derive a finite number of formulas F satisfying both the following conditions :

- (i) $H \rightarrow F \rightarrow W$
- (ii) F is more general than W .

Of course, if H is more general than W , then we could get $F = H$. We will show that this method is complete, i.e. that for any two formulas H and W , it is possible to derive F from H by our method such that (i) and (ii) above hold. The algorithms given will involve some nondeterminism.

Let H, W be first-order formulas such that $H \rightarrow W$. In Section 2.2 we will present an algorithm for unskolemizing a set of clauses \mathcal{D} derived from $Sk(H)$. The properties of the unskolemization algorithm will be discussed in Section 2.3.

Throughout this section, we will keep illustrating the concepts introduced with the help of an example, which will be taken through the section to demonstrate the working of the theory. The formulas used in the example will be :

$$H = \forall x \forall y \forall z \forall w ((Q(y) \vee L(b, y)) \wedge \neg Q(g(a)) \wedge L(g(a), a) \wedge (R(x, g(a)) \vee \neg P(x, g(a))) \wedge (\neg R(w, z) \vee \neg D(w, z)))$$

$$W = \exists u \forall v (L(b, u) \wedge L(u, a) \wedge (\neg P(v, u) \vee \neg D(v, u) \vee M(a))).$$

It can be proved that $H \rightarrow W$. Now, W is given here only to show that we will ultimately be able to derive a formula F such that $H \rightarrow F \rightarrow W$ and such that F is more general than W if we make the correct choices whenever nondeterminism is involved. In actual practice, W will be unknown, and the choices which we will make in the examples based on properties of W will have to be made nondeterministically.

2.2 The unskolemization process

2.2.1 Preliminaries

Unskolemization has been defined by some authors as the process of eliminating Skolem functions from a formula without quantifiers, replacing them with new existentially quantified variables, and transforming the resulting formula into a closed formula with quantifiers. Some unskolemization algorithms have been developed in the past. McCune [McCune 88] presents an algorithm to solve the following problem : given a set S of clauses and a set F of constant and function symbols that occur in the clauses of S , obtain a fully quantified (closed) formula S' from S by replacing expressions starting with symbols in F with existentially quantified variables. S' is unsatisfiable if and only if S is unsatisfiable. The algorithm which is presented is sound but not complete. The following approach is used by McCune for handling the case in which a function symbol to be eliminated has a non-variable argument or more than one occurrence of an argument. Rewrite the clause, replacing the non-variable argument, say t , with a new variable, say x ; then append a new literal $x \neq t$ to the clause. For the unskolemization to be successful, every occurrence of a Skolem symbol to be eliminated must have the same sequence of arguments. Also, for $m \leq n$, if f is an m -ary function symbol and g is an n -ary function symbol, the arguments of f must be a subset of the arguments of g . Thus the unskolemization algorithm is not complete since it fails on any set of clauses not satisfying the above criteria.

Cox and Pietrzykowski [Cox and Pietrzykowski 84] present an algorithm for unskolemization, but their algorithm is applicable only to literals, and their goal

is to produce a set of quantified atomic formulas, each of which conflicts with the given literal.

Before plunging into the details of the unskolemization process, we should remark that the term “unskolemization” is used rather loosely here. As mentioned above, other authors ([Cox and Pietrzykowski 84], [McCune 88]) regard unskolemization as the process of eliminating Skolem functions from a formula without quantifiers, replacing them with new existentially quantified variables, and transforming the resulting formula into a closed formula with quantifiers. We will expand the meaning of unskolemization slightly. In our definition, ordinary function symbols can also be “unskolemized” by treating them as if they were Skolem functions. Thus a function symbol may be replaced by an existentially quantified variable during the unskolemization process. To illustrate this, suppose we want to unskolemize the formula

$$P(f(x)) \vee Q(g(a), x)$$

where f and a are (non-Skolem) function symbols, and suppose we want to treat f and a as if they were Skolem functions. The resulting formula would be

$$\exists z \forall x \exists y (P(y) \vee Q(g(z), x)).$$

Note that if we Skolemize $\exists z \forall x \exists y (P(y) \vee Q(g(z), x))$, we will get back the original formula (up to names of Skolem functions). In practice, the situation may be a little more complicated as the formula being unskolemized may not be the Skolemized form of any formula. The given algorithm shows how to cope with such situations.

Also, unskolemization as presented here does not necessarily preserve unsatisfiability. For example, the formula

$$\exists x (\text{succ}(x) = 0)$$

is false under the usual interpretation of “succ” as the successor function over natural numbers; however, if we unskolemize this function we get the formula

$$\exists y (y = 0)$$

which is true. The properties of our unskolemization algorithm will be detailed in Section 2.3.

We motivate the development of the unskolemization algorithm by the following simple example. Let A be a formula which implies some formula B , which is unknown. Since $A \rightarrow B$, $A \wedge \neg B$ is unsatisfiable; therefore $Sk(A \wedge \neg B)$ is unsatisfiable (since Skolemization preserves unsatisfiability), i.e. $Sk(A) \wedge Sk(\neg B)$ is unsatisfiable. Thus some set of clauses \mathcal{D} can be derived from A such that $\mathcal{D} \wedge Sk(\neg B)$ is unsatisfiable. It may happen that some literals in \mathcal{D} are instances of some literals in B . For example, suppose that $B = \exists x P(x)$, and suppose $A = P(a)$, where a is a ground

term. Clearly $A \rightarrow B$. Now, in order to derive B from the formula $A = P(a)$, the ground term a has to be “unskolemized” by replacing a by an existential quantifier. If this is done for $P(a)$, the resulting formula will be $\exists xP(x)$.

The above example is very simple and straightforward; in actual practice, there may be many function symbols in A , some of which we may need to replace by existential quantifiers and some which should not be thus replaced. This, then, explains why our unskolemization algorithm is nondeterministic.

Consider an unknown formula B and some formula A which implies B . As explained above, $Sk(A) \wedge Sk(\neg B)$ is unsatisfiable. Therefore by the completeness of resolution, we can derive the empty clause from $Sk(A) \wedge Sk(\neg B)$. Now, B is unknown, and we want to derive it from $Sk(A)$. It may not be possible to derive B from $Sk(A)$ without using tautologies or unskolemization, as is demonstrated by the two following examples :

(i) Suppose $A = P$, $B = P \vee Q \vee R$.

Clearly $A \rightarrow B$.

But the only way to derive B from A by resolution is by resolving A with the tautology $\{\neg P, P, Q, R\}$. However, the need for resolving with tautologies would increase the size of the search space tremendously, since there exist an infinite number of such tautologies. Thus the use of tautologies is best avoided.

(ii) Suppose $A = P(a)$, $B = \exists xP(x)$. Then B (or even $Sk(B)$) cannot be derived from A by resolution. Obtaining B from A requires replacing “ a ” by an existentially quantified variable, as was just shown above.

In conclusion, although B can be derived by resolution and unskolemization from $Sk(A)$ (by the completeness of the resolution principle), such a derivation entails the use of tautologies during the resolution process. Also, it is unclear how to handle the unskolemization without a formal algorithm for doing so. This is best illustrated by an example :

Suppose $A = \forall x \forall z P(x, f(a, x), z, g(z))$, $B = \forall x \exists y \forall z \exists w P(x, y, z, w)$.

Clearly $A \rightarrow B$. To obtain B from A , we need to replace the terms $f(a, x)$ and $g(z)$ in A by existentially quantified variables. Suppose $f(a, x)$ and $g(z)$ are replaced by existentially quantified variables x_1 and z_1 respectively. The question remains about where to place the existential quantifiers $\exists x_1$ and $\exists z_1$ in the quantifier string for A . Since $f(a, x)$ was replaced by x_1 , $\exists x_1$ should come after $\forall x$ (since a is a ground term, its presence as an argument of f is inconsequential); similarly, since $g(z)$ was replaced by z_1 , $\exists z_1$ should come after $\forall z$. There are thus two choices for the unskolemized version of A , namely

$\forall x \exists x_1 \forall z \exists z_1 P(x, x_1, z, z_1)$, and

$$\forall z \exists z_1 \forall x \exists x_1 P(x, x_1, z, z_1),$$

of which the former is the correct choice in this case.

In order to address the above issues formally, we present an unskolemization algorithm. Suppose we have a formula H which implies W , where W is an unknown formula that we want to find. We shall show how to unskolemize a set of clauses \mathcal{D} to give a family of formulas \mathcal{K} . Briefly, the objective of unskolemizing \mathcal{D} is to replace function symbols of \mathcal{D} which do not occur in W by existentially quantified variables, where \mathcal{D} is a set of clauses derived from $Sk(H)$ by resolution. That is, if for some literal $D'_{j,k}$ in \mathcal{D} , an argument d_i of $D'_{j,k}$ is a function symbol which does not appear in W , then we will unskolemize the function symbol of d_i during the unskolemization of the set of clauses \mathcal{D} to get a family of new formulas \mathcal{K} . Thus any $F \in \mathcal{K}$ will contain a new existentially quantified variable in place of d_i . Since W is unknown, this procedure will have to be carried out nondeterministically. This process will make the unskolemized formula “more general” than W (this term will be defined later).

We will unskolemize \mathcal{D} according to an algorithm given in the next section and obtain a family of formulas \mathcal{L} . For each formula $F \in \mathcal{K}$, where \mathcal{K} is a subset of \mathcal{L} to be defined later, we will show that F is “more general” than W according to a definition given later. Also, we show that there exists $\mathbf{F} \in \mathcal{K}$ such that $\mathbf{F} \rightarrow W$. We also show that $H \rightarrow F$ for all $F \in \mathcal{L}$; hence $H \rightarrow \mathbf{F} \rightarrow W$. Finally, we will define a relation “ \preceq ” and show that $\mathbf{F} \preceq W$, and that $\{F \mid F \preceq W\}$ is finite up to variants under certain syntactic constraints.

Note. The following algorithm makes use of the *guarded command* for conditional statements [Gries 81]. Briefly, the general form of a conditional statement is

```

if  $B_1 \rightarrow S_1$ 
  []  $B_2 \rightarrow S_2$ 
  ...
  []  $B_n \rightarrow S_n$ 
fi

```

where $n \geq 0$ and each $B_i \rightarrow S_i$ is a guarded command. Each S_i can be any statement. The command is executed as follows. First, if any guard B_i is not well-defined in the state in which execution begins, abortion may occur. Second, if none of the guards is true, then abortion occurs; and finally, if at least one guard is true, then one guarded command $B_i \rightarrow S_i$ with true guard B_i is chosen and S_i executed. Note that if more than one guard is true, then one of the guarded commands $B_i \rightarrow S_i$ with true guard B_i is chosen arbitrarily and S_i is executed. Thus the execution of such a statement can be nondeterministic. Notice that in

steps 2 and 3 of the following algorithm, two of the guards are identical. This serves as a convenient way of representing nondeterminism; the meaning here is that if the two identical guards are true in one of the steps, then one of the actions specified will be performed and this action will be picked arbitrarily from the two available actions.

2.2.2 The unskolemization algorithm

INPUT : A set "*CLAUSES*" of clauses to be unskolemized.

Step 1. Make i_k copies of every clause C_k of *CLAUSES*, where i_k is some integer (chosen nondeterministically), and rename variables in all clauses so that no two clauses have any variable in common. Call the resulting set of clauses *MULTIPLE_CLAUSES*.

Comment : In actual practice, for each k , we can try setting i_k to 1, then to 2, then to 3, and so on. Eventually i_k will become large enough. This is because it is possible to bound i_k by the number of resolutions performed when deriving the empty clause from $Sk(H \wedge \neg W)$. The reason for this is demonstrated in the proof of Theorem 2.1.

Step 2. For every literal L in every clause of *MULTIPLE_CLAUSES*, process the arguments of L as follows. Suppose $L = \pm P(d_1, d_2, \dots, d_s)$ (where \pm designates the sign of the literal L). For $i = 1$ to s do

if (d_i is a term beginning with a Skolem function symbol) \rightarrow
 replace d_i by $X \leftarrow d_i$, for some variable X not occurring anywhere else in any other clause;

[] (d_i is a term beginning with a non-Skolem function symbol) \rightarrow
 replace d_i by $X \leftarrow d_i$, for some variable X not occurring anywhere else in any other clause;

[] (d_i is a term beginning with a non-Skolem function symbol) \rightarrow
 skip;

[] (d_i is a term which begins with neither a Skolem function symbol nor a non-Skolem function symbol) \rightarrow
 skip

fi

Call the resulting formula *MARK*.

Step 3. For every pair of marked arguments " $X \leftarrow \alpha$ ", " $Y \leftarrow \beta$ " in *MARK* do

if α, β are unifiable \rightarrow

 replace all occurrences of X and Y by a new variable Z ;

\square α, β are unifiable \rightarrow
 skip;
 \square α, β are not unifiable \rightarrow
 skip
 fi

If any two clauses become identical, drop one of them.

Step 4. For every argument of the form " $X \leftarrow f(\nu_1, \nu_2, \dots, \nu_r)$ " in *MARK* (where $r \geq 1$) do

 replace " $X \leftarrow f(\nu_1, \nu_2, \dots, \nu_r)$ " by " $X \leftarrow f(y_1, y_2, \dots, y_n)$ ", where y_1, \dots, y_n are all the distinct variables occurring in $\nu_1, \nu_2, \dots, \nu_r$.

Comment : Note that we are not changing the arity of function symbols here. Dropping arguments is done purely for computational purposes and has no bearing on the arity of the function.

Step 5. Add universal quantifiers for all the free variables at the head of the formula.

Step 6. Unskolemize the remaining marked arguments of the formula as follows. Let **A** and **C** be two sets which are initially empty. Collect together all marked arguments with the same variable on the left-hand side of the " \leftarrow " sign. Suppose these are

$$x \leftarrow \alpha_1, x \leftarrow \alpha_2, \dots, x \leftarrow \alpha_n.$$

Let y_1, y_2, \dots, y_r be all the variables occurring in $\alpha_1, \alpha_2, \dots, \alpha_n$. Then replace " $x \leftarrow \alpha_i$ ", for $1 \leq i \leq n$, everywhere by a new variable, say z , and add the r ordered pairs (y_i, z) to **C**. If $r = 0$, then add z to the set **A**.

Comment : **C** is the set of constraints on the ordering of new existential quantifiers, relative to universal quantifiers which were introduced in Step 5. The presence of an ordered pair (y_i, z) in **C** signifies that " $\exists z$ " must come *after* " $\forall y_i$ " in the quantifier string of the unskolemized formula. **A** is the set of existentially quantified variables which do not depend on any other variables and whose existential quantifiers can precede all the universal quantifiers of the unskolemized formula.

Step 7. Complete the quantifier string of the unskolemized formula. From Step 5, we already have a partially constructed quantifier string. We complete this string using the sets **A** and **C** as follows :

(i) Add " $\exists x$ " at the head of the partially completed quantifier string for every $x \in \mathbf{A}$ (in any order).

(ii) Gather all ordered pairs which have the same second component into groups and process each group as follows. Suppose the r ordered pairs $(y_i, z), 1 \leq i \leq r$ occur in one of these groups. Then add " $\exists z$ " to the quantifier string so that " $\exists z$ "

comes after “ $\forall y_i$ ” for every $i, 1 \leq i \leq r$, rearranging the string as specified below if necessary.

While performing the above steps, it is allowed to rearrange the order of any string of consecutive universal quantifiers and of any string of consecutive existential quantifiers, if desired. We will rearrange this order so that the resulting formula is as strong as possible. Rearrange these so that the constraints are satisfied optimally. By “optimally” we mean that if for example the r ordered pairs $(y_i, z), 1 \leq i \leq r$ are all the ordered pairs which have the same second component z in \mathbf{C} , then “ $\exists z$ ” is added to the quantifier string so that “ $\exists z$ ” comes after “ $\forall y_i$ ” for every $i, 1 \leq i \leq r$; in addition, “ $\exists z$ ” must come after as few other universal quantifiers as possible. In other words, “ $\exists z$ ” must be placed as much to the front of the quantifier string as possible without violating the constraints of \mathbf{C} . In case there is a choice involved, e.g. if $\mathbf{C} = \{(x, y), (z, w)\}$, then there are two possible “optimal” quantifier strings, viz. “ $\forall x \exists y \forall z \exists w$ ” and “ $\forall z \exists w \forall x \exists y$ ”. In other words, there may be more than one way of rearranging the quantifiers optimally. •

Notes

1. Since the ordered pairs of \mathbf{C} can be processed in any order, it is possible to obtain many different quantifier strings by using the above method. However, the relative order of the universal and existential quantifiers obtained at each step cannot be altered.

2. The set \mathbf{C} imposes a partial order on the existential and universal quantifiers of the formula; however, any first-order formula must have a total ordering on its existential and universal quantifiers. Many total orders may be constructed which satisfy the partial order imposed by the set \mathbf{C} . We choose the orders which will make the resulting formula as strong as possible.

It is thus possible to obtain many different formulas when unskolemizing a given set of clauses; in other words, given a set of clauses A , $unsk(A)$ obtained by using this algorithm is not necessarily a singleton, where $unsk(A)$ denotes the set of formulas obtained by unskolemizing A . We will however choose one such formula \mathbf{F} such that $H \rightarrow \mathbf{F} \rightarrow W$. The proof that \mathbf{F} exists is given in the next section. Also, if some set of clauses A is unskolemized, then $Sk(unsk(A))$ is not necessarily equal to A .

2.3 Analysis of the unskolemization algorithm

In this section, a number of theorems are proved and certain concepts are defined. The theorems will serve to characterize the properties of formulas obtained

by unskolemizing a set of clauses using the unskolemization algorithm of the previous section. They will also be used to prove the soundness and completeness of an algorithm for deriving loop invariants in Chapter 3 and the soundness of a learning algorithm in Chapter 4.

Lemma 2.1 Let H, W be two first-order formulas such that $H \rightarrow W$. Then there exists a set T' of ground instances of clauses of $T = Sk(\neg W)$, and there exists a set of clauses \mathcal{D} derivable by resolution from $S = Sk(H)$ such that for each clause D_i of $\neg T'$, there exists a clause D'_i of \mathcal{D} such that D'_i subsumes D_i (i.e. there exists a substitution θ_i such that $D'_i\theta_i \subseteq D_i$).

Proof : Let $S = Sk(H)$, $T = Sk(\neg W)$. Since $H \rightarrow W$, we know that $S \wedge T$ is unsatisfiable. Therefore by Herbrand's theorem, there exists a set T' of clauses which are ground instances of clauses of T such that $S \wedge T'$ is unsatisfiable; hence $H \rightarrow \neg T'$. Express $\neg T'$ in conjunctive normal form as

$$\neg T' = D_1 \wedge D_2 \wedge \dots \wedge D_n, \text{ say, where each } D_i \text{ is a disjunction of literals.}$$

Since $H \rightarrow D_1 \wedge D_2 \wedge \dots \wedge D_n$, we have

$$H \rightarrow D_i \text{ for all } i, 1 \leq i \leq n.$$

Here, D_i is a ground clause and $H \rightarrow D_i$, therefore $S \wedge \neg D_i$ is unsatisfiable and the empty clause can be derived by resolution from $S \wedge \neg D_i$. Now, $\neg D_i$ is a set of unit clauses. First perform any required resolutions among clauses of S . Then use the clauses from $\neg D_i$. Each resolution using a clause from $\neg D_i$ has for effect the removal of a literal from some clause from $Res(S)$, along with a possible instantiation. Therefore clearly we need to resolve the unit clauses of $\neg D_i$ only against one clause from $Res(S)$. But then this clause must subsume D_i . Therefore for each D_i , there exists $D'_i \in Res(S)$, where $Res(S)$ is the set of all possible resolvents of S , such that D'_i subsumes D_i (i.e. there exists a substitution θ_i such that $D'_i\theta_i \subseteq D_i$). •

Lemma 2.2 Using the notation of Lemma 2.1, $(D'_1 \wedge D'_2 \wedge \dots \wedge D'_n) \wedge T$ is unsatisfiable.

Proof : Let $\mathcal{D} = D'_1 \wedge D'_2 \wedge \dots \wedge D'_n$, $\neg T' = D_1 \wedge D_2 \wedge \dots \wedge D_n$. From Lemma 2.1, there exist substitutions $\theta_1, \theta_2, \dots, \theta_n$ such that

$$D'_i\theta_i \subseteq D_i \text{ for each } i.$$

Therefore

$$D'_i\theta_i \rightarrow D_i \text{ for all } i \text{ (regarding free variables as universally quantified).}$$

Hence

$$(D'_1\theta_1 \wedge D'_2\theta_2 \wedge \dots \wedge D'_n\theta_n) \rightarrow D_1 \wedge D_2 \wedge \dots \wedge D_n.$$

Therefore

$$(D'_1\theta_1 \wedge D'_2\theta_2 \wedge \dots \wedge D'_n\theta_n) \wedge \neg(D_1 \wedge D_2 \wedge \dots \wedge D_n) \text{ is unsatisfiable, therefore}$$

$(D'_1 \wedge D'_2 \wedge \dots \wedge D'_n) \wedge \neg (D_1 \wedge D_2 \wedge \dots \wedge D_n)$ is also unsatisfiable (since the $D'_i\theta_i$'s are instances of the D_i 's).

i.e. $(D'_1 \wedge D'_2 \wedge \dots \wedge D'_n) \wedge T'$ is unsatisfiable.

Therefore $(D'_1 \wedge D'_2 \wedge \dots \wedge D'_n) \wedge T$ is unsatisfiable (since T implies T')

i.e. $\mathcal{D} \wedge T$ is unsatisfiable. •

Example 2.1 Using the formulas H and W given at the end of Section 2.1, we have

$$Sk(H) = \{\{Q(y), L(b, y)\}, \{\neg Q(g(a))\}, \{L(g(a), a)\}, \{R(x, g(a)), \neg P(x, g(a))\}, \{\neg R(w, z), \neg D(w, z)\}\}$$

$$\neg W = \forall u \exists v ((\neg L(b, u) \vee \neg L(u, a) \vee P(v, u)) \wedge (\neg L(b, u) \vee \neg L(u, a) \vee D(v, u)) \wedge (\neg L(b, u) \vee \neg L(u, a) \vee \neg M(a)))$$

$$T = Sk(\neg W) = \{\{\neg L(b, u), \neg L(u, a), P(f(u), u)\}, \{\neg L(b, u), \neg L(u, a), D(f(u), u)\}, \{\neg L(b, u), \neg L(u, a), \neg M(a)\}\}.$$

In $Sk(\neg W)$, f is a Skolem function replacing the existentially quantified variable “ v ” of $\neg W$. A set T' of clauses which are ground instances of clauses of T such that $S \wedge T'$ is unsatisfiable is :

$$T' = \{\{\neg L(b, g(a)), \neg L(g(a), a), P(f(g(a)), g(a))\}, \{\neg L(b, g(a)), \neg L(g(a), a), D(f(g(a)), g(a))\}\}.$$

Therefore $\neg T' = \{\{L(b, g(a))\}, \{L(g(a), a)\}, \{\neg P(f(g(a)), g(a)), \neg D(f(g(a)), g(a))\}\}.$

Write

$$D_1 = L(b, g(a)),$$

$$D_2 = L(g(a), a),$$

$$D_3 = \neg P(f(g(a)), g(a)) \vee \neg D(f(g(a)), g(a));$$

then $\neg T' = D_1 \wedge D_2 \wedge D_3$. Also,

$$H \rightarrow D_1, H \rightarrow D_2, H \rightarrow D_3.$$

The five clauses of $Sk(H)$ are listed below :

1. $\{Q(y), L(b, y)\}$
2. $\{\neg Q(g(a))\}$
3. $\{L(g(a), a)\}$
4. $\{R(x, g(a)), \neg P(x, g(a))\}$
5. $\{\neg R(w, z), \neg D(w, z)\}$

Perform the following resolutions among these clauses to get the following clauses :

6. $\{L(b, g(a))\}$ from clauses 1 and 2
7. $\{\neg P(x, g(a)), \neg D(x, g(a))\}$ from clauses 4 and 5.

The clauses $D'_1, D'_2, D'_3 \in Res(Sk(H))$ which subsume D_1, D_2 , and D_3 respectively are :

$$\begin{aligned} D'_1 &= \{L(b, g(a))\} && \text{(clause 6 above),} \\ D'_2 &= \{L(g(a), a)\} && \text{(clause 3 above),} \\ D'_3 &= \{\neg P(x, g(a)), \neg D(x, g(a))\} && \text{(clause 7 above).} \end{aligned}$$

Here $D_1 = D'_1, D_2 = D'_2, D_3 = D'_3\theta_3$, where $\theta_3 = \{x \leftarrow f(g(a))\}$.

Note : $\mathcal{D} = D'_1 \wedge D'_2 \wedge D'_3$. Also note that $\mathcal{D} \wedge T$ is unsatisfiable. •

Lemma 2.3 Using the same notation as in Lemmas 2.1 and 2.2, for any literal L of \mathcal{D} , where $L = \pm P(d_1, d_2, \dots, d_s)$, say (\pm denotes the sign of L), there exists a literal M of W such that $M = \pm P(b_1, b_2, \dots, b_s)$ (M has the same sign as L) and such that for each i , $1 \leq i \leq s$, the following holds :

- (i) If d_i is a Skolem function (with zero or more arguments), then b_i is a variable which is existentially quantified in W .
- (ii) If d_i is a non-Skolem function (with zero or more arguments), then one of the following holds :
 - (a) b_i is the same function symbol with the same number of arguments, and (i) and (ii) here hold recursively for each corresponding argument of d_i and b_i .
 - (b) b_i is an existentially quantified variable and the function symbol of d_i (with the same arity as d_i) appears in W .
 - (c) b_i is an existentially quantified variable in W and the function symbol of d_i does not appear anywhere in W .

Proof : Let

$\mathcal{D} = D'_1 \wedge D'_2 \wedge \dots \wedge D'_n$, where the D'_i 's are clauses,

$D'_j = D'_{j1} \vee D'_{j2} \vee \dots \vee D'_{ji}$ for all $1 \leq j \leq n$, where the D'_{jk} 's are literals,

$W = Q^W(V_1 \wedge V_2 \wedge \dots \wedge V_m)$, where W is expressed in conjunctive normal form, the V_i 's are disjunctions of literals, and Q^W is the quantifier string of W ; hence

$\neg W = (\neg Q^W)(\neg V_1 \vee \neg V_2 \vee \dots \vee \neg V_m)$. Write

$(\neg V_1 \vee \neg V_2 \vee \dots \vee \neg V_m)$ in conjunctive normal form; suppose

$(\neg V_1 \vee \neg V_2 \vee \dots \vee \neg V_m) \equiv (W'_1 \wedge W'_2 \wedge \dots \wedge W'_p)$. Let σ be the substitution

which replaces existentially quantified variables of $\neg W$ by Skolem symbols to get $Sk(\neg W)$, i.e.

$$\begin{aligned} &(\neg V_1 \vee \neg V_2 \vee \dots \vee \neg V_m)\sigma \\ &= (W'_1 \wedge W'_2 \wedge \dots \wedge W'_p)\sigma \\ &= Sk(\neg W). \end{aligned}$$

Therefore $Sk(\neg W) = W'_1\sigma \wedge W'_2\sigma \wedge \dots \wedge W'_p\sigma$.

Suppose $W'_k = W'_{k1} \vee W'_{k2} \vee \dots \vee W'_{kj_k}$, $\forall 1 \leq k \leq p$. Then

$$\begin{aligned}
W &= Q^W(V_1 \wedge V_2 \wedge \dots \wedge V_m) \\
&\equiv Q^W(\neg W'_1 \vee \neg W'_2 \vee \dots \vee \neg W'_p) \\
&\equiv Q^W((\neg W'_{11} \wedge \neg W'_{12} \wedge \dots \wedge W'_{1j_1}) \vee (\neg W'_{21} \wedge \neg W'_{22} \wedge \dots \wedge \neg W'_{2j_2}) \vee \dots \vee (\neg W'_{p1} \wedge \neg W'_{p2} \wedge \dots \wedge \neg W'_{pj_p})).
\end{aligned}$$

Now, T' is a set of ground instances of clauses of $Sk(\neg W)$ (using the same notation as in the previous two lemmas), therefore

$T' = W'_{k_1} \sigma \alpha_1 \wedge W'_{k_2} \sigma \alpha_2 \wedge \dots \wedge W'_{k_m} \sigma \alpha_m$, where $1 \leq k_j \leq p$ for each j such that $1 \leq j \leq m$, and where each α_j instantiates the corresponding $W'_{k_j} \sigma$ into a ground clause. Therefore

$$\begin{aligned}
\neg T' &= \neg W'_{k_1} \sigma \alpha_1 \vee \neg W'_{k_2} \sigma \alpha_2 \vee \dots \vee \neg W'_{k_m} \sigma \alpha_m \\
&\stackrel{=}{=} \bigwedge_{\substack{1 \leq a_i \leq j_{k_i} \\ 1 \leq i \leq m}} (\neg W'_{k_1 a_1} \sigma \alpha_1 \vee \neg W'_{k_2 a_2} \sigma \alpha_2 \vee \dots \vee \neg W'_{k_m a_m} \sigma \alpha_m),
\end{aligned}$$

where each k_i is such that $1 \leq k_i \leq p$.

But recall that in Lemma 2.1, we had written

$$\neg T' = D_1 \wedge D_2 \wedge \dots \wedge D_n,$$

and had obtained the result that for each l such that $1 \leq l \leq n$, there exists θ_l such that $D'_l \theta_l \subseteq D_l$. Hence

$$\bigwedge_{\substack{1 \leq a_i \leq j_{k_i} \\ 1 \leq i \leq m}} (\neg W'_{k_1 a_1} \sigma \alpha_1 \vee \neg W'_{k_2 a_2} \sigma \alpha_2 \vee \dots \vee \neg W'_{k_m a_m} \sigma \alpha_m) \equiv D_1 \wedge D_2 \wedge \dots \wedge D_n.$$

Therefore for each l such that $1 \leq l \leq n$, there exists a clause $(\neg W'_{k_1 a_1} \sigma \alpha_1 \vee \neg W'_{k_2 a_2} \sigma \alpha_2 \vee \dots \vee \neg W'_{k_m a_m} \sigma \alpha_m)$, where $1 \leq a_i \leq j_{k_i}$, $1 \leq k_i \leq p$, $1 \leq i \leq m$ such that

$$\begin{aligned}
D'_l \theta_l &\subseteq D_l \\
&= \{\neg W'_{k_1 a_1} \sigma \alpha_1, \neg W'_{k_2 a_2} \sigma \alpha_2, \dots, \neg W'_{k_m a_m} \sigma \alpha_m\}.
\end{aligned}$$

Therefore given any D'_l , there exists a substitution θ_l such that $D'_l \theta_l \subseteq D_l$, and this D_l can be written as a disjunction of some instances of literals of W . Now,

$$\begin{aligned}
D'_j &= \{D'_{j_1}, D'_{j_2}, \dots, D'_{j_{i_j}}\}; \text{ therefore} \\
D'_j \theta_j &= \{D'_{j_1} \theta_j, D'_{j_2} \theta_j, \dots, D'_{j_{i_j}} \theta_j\} \subseteq \{\neg W'_{k_1 a_1} \sigma \alpha_1, \neg W'_{k_2 a_2} \sigma \alpha_2, \dots, \neg W'_{k_m a_m} \sigma \alpha_m\}.
\end{aligned}$$

This means that for each literal D'_{jk} of D'_j , there exists a literal $\neg W'_{k_l a_l}$ of W such that

$$D'_{jk} \theta_j = \neg W'_{k_l a_l} \sigma \alpha_l$$

where σ is the substitution which replaces existentially quantified variables of $\neg W$ by Skolem symbols to get $Sk(\neg W)$ (i.e. $\neg W \sigma = Sk(\neg W)$), and α_l is a substitution which makes $\neg W'_{k_l a_l} \sigma \alpha_l$ a ground literal. Note that $W'_{k_l a_l} \sigma$ is basically nothing other than the literal against which the literal D'_{jk} is resolved during the derivation of the empty clause from $\mathcal{D} \wedge Sk(\neg W)$. Also note that since σ replaces all universally

quantified variables in W by new Skolem symbols, all the variables of $\neg W'_{k_1 a_1} \sigma$ are existentially quantified in W .

Now suppose $D'_{jk} = \pm P(d_1, d_2, \dots, d_s)$ for some predicate P , where \pm is the sign of the literal D'_{jk} ; then

$\neg W'_{k_1 a_1} = \pm P(b_1, b_2, \dots, b_s)$ and $\neg W'_{k_1 a_1} \sigma = \pm P(c_1, c_2, \dots, c_s)$, say, for some arguments $b_i, c_i, 1 \leq i \leq s$, and the sign of these literals is the same as the sign of D'_{jk} . For any i such that $1 \leq i \leq s$, if d_i is a variable, b_i could be anything. If d_i is not a variable, then either of the following could hold :

Case (i) : If d_i is a Skolem symbol (with zero or more arguments), then (since b_i cannot contain the same Skolem symbol) b_i must be a variable. If this variable were universally quantified in W , then c_i would be a new Skolem symbol and therefore could not unify with d_i ; hence b_i must be an existentially quantified variable in W .

Case (ii) : If d_i is a function symbol (with zero or more arguments), then one of the following are possible :

(a) b_i is the same function symbol (with the same number of arguments).

(b) b_i is a variable and the function symbol of d_i (with the same arity as d_i) appears in W . By the same argument as in (i) above, b_i must be existentially quantified in W .

(c) b_i is a variable and the function symbol of d_i (with the same arity as d_i) does not appear anywhere in W . By the same argument as in (i) above, b_i must be existentially quantified in W .

For Case (ii) (a), if the function symbol which is common to d_i and b_i has more than zero arguments, repeat the above analysis recursively for all these arguments (this analysis must eventually terminate since D'_{jk} and $\neg W'_{k_1 a_1}$ are of finite length).

Since every literal of \mathcal{D} can be written as D'_{jk} for some $1 \leq j \leq n, 1 \leq k \leq i_j$, and since the corresponding $\neg W'_{k_1 a_1}$ is a literal of W (corresponding to M in the statement of the lemma), the lemma follows. •

Example 2.2 Continuing with the results in Example 2.1, and using the notation of the previous lemmas, we illustrate the results in this section. We have,

$$D'_1 = \{L(b, g(a))\} = \{D'_{11}\},$$

$$D'_2 = \{L(g(a), a)\} = \{D'_{21}\},$$

$$D'_3 = \{\neg P(x, g(a)), \neg D(x, g(a))\} = \{D'_{31}, D'_{32}\}.$$

Now, $\neg W = \forall u \exists v ((\neg L(b, u) \vee \neg L(u, a) \vee P(v, u)) \wedge (\neg L(b, u) \vee \neg L(u, a) \vee D(v, u)) \wedge (\neg L(b, u) \vee \neg L(u, a) \vee \neg M(a)))$. σ replaced the existentially quantified variable v in $\neg W$ by $f(u)$; therefore $\sigma = \{v \leftarrow f(u)\}$. We write

$$\neg W = \forall u \exists v (W'_1 \wedge W'_2 \wedge W'_3)$$

$$= \forall u \exists v ((W'_{11} \vee W'_{12} \vee W'_{13}) \wedge (W'_{21} \vee W'_{22} \vee W'_{23}) \wedge (W'_{31} \vee W'_{32} \vee W'_{33})), \text{ where}$$

$$\begin{array}{lll}
W'_{11} = \neg L(b, u) & W'_{21} = \neg L(b, u) & W'_{31} = \neg L(b, u) \\
W'_{12} = \neg L(u, a) & W'_{22} = \neg L(u, a) & W'_{32} = \neg L(u, a) \\
W'_{13} = P(v, u) & W'_{23} = D(v, u) & W'_{33} = \neg M(a)
\end{array}$$

and $Sk(\neg W) = (W'_{11}\sigma \vee W'_{12}\sigma \vee W'_{13}\sigma) \wedge (W'_{21}\sigma \vee W'_{22}\sigma \vee W'_{23}\sigma) \wedge (W'_{31}\sigma \vee W'_{32}\sigma \vee W'_{33}\sigma)$.

Recall that

$$\begin{aligned}
T' &= \{ \{ \neg L(b, g(a)), \neg L(g(a), a), P(f(g(a)), g(a)) \}, \\
&\quad \{ \neg L(b, g(a)), \neg L(g(a), a), D(f(g(a)), g(a)) \} \} \\
&= W'_1\sigma\alpha_1 \wedge W'_2\sigma\alpha_1
\end{aligned}$$

where α_1 is the substitution $\{u \leftarrow g(a)\}$ which transforms clauses of T into ground clauses in T' . Therefore

$$\begin{aligned}
\neg T' &= \neg W'_1\sigma\alpha_1 \vee \neg W'_2\sigma\alpha_1 \\
&= (\neg W'_{11}\sigma\alpha_1 \wedge \neg W'_{12}\sigma\alpha_1 \wedge \neg W'_{13}\sigma\alpha_1) \vee (\neg W'_{21}\sigma\alpha_1 \wedge \neg W'_{22}\sigma\alpha_1 \wedge \neg W'_{23}\sigma\alpha_1) \\
&= (\neg W'_{11}\sigma\alpha_1 \vee \neg W'_{21}\sigma\alpha_1) \wedge (\neg W'_{11}\sigma\alpha_1 \vee \neg W'_{22}\sigma\alpha_1) \wedge (\neg W'_{11}\sigma\alpha_1 \vee \neg W'_{23}\sigma\alpha_1) \wedge \\
&\quad (\neg W'_{12}\sigma\alpha_1 \vee \neg W'_{21}\sigma\alpha_1) \wedge (\neg W'_{12}\sigma\alpha_1 \vee \neg W'_{22}\sigma\alpha_1) \wedge (\neg W'_{12}\sigma\alpha_1 \vee \neg W'_{23}\sigma\alpha_1) \wedge \\
&\quad (\neg W'_{13}\sigma\alpha_1 \vee \neg W'_{21}\sigma\alpha_1) \wedge (\neg W'_{13}\sigma\alpha_1 \vee \neg W'_{22}\sigma\alpha_1) \wedge (\neg W'_{13}\sigma\alpha_1 \vee \neg W'_{23}\sigma\alpha_1) \\
&\equiv L(b, g(a)) \wedge L(g(a), a) \wedge (\neg P(f(g(a)), g(a)) \vee \neg D(f(g(a)), g(a)))
\end{aligned}$$

(which is what we had obtained before for $\neg T'$). Also,

$$\neg T' \equiv D'_1 \wedge D'_2 \wedge D'_3\theta_3.$$

So we have,

$$\begin{aligned}
D'_{11} &= \neg W'_{11}\sigma\alpha_1 \\
D'_{21} &= \neg W'_{12}\sigma\alpha_1 \\
D'_{31}\theta_3 &= \neg W'_{13}\sigma\alpha_1; \quad D'_{32}\theta_3 = \neg W'_{23}\sigma\alpha_1.
\end{aligned}$$

We now perform the case analysis as in Lemma 2.3 for each literal of \mathcal{D} :

$$\begin{aligned}
\bullet D'_{11} &= L(b, g(a)) = \neg W'_{11}\sigma\alpha_1 \\
&\quad \neg W'_{11} = L(b, u).
\end{aligned}$$

For the first argument of D'_{11} , which is “ b ”, case (ii) (a) applies.

For the second argument of D'_{11} , which is “ $g(a)$ ”, case (ii) (c) applies.

$$\begin{aligned}
\bullet D'_{21} &= L(g(a), a) = \neg W'_{12}\sigma\alpha_1 \\
&\quad \neg W'_{12} = L(u, a).
\end{aligned}$$

For the first argument of D'_{21} , which is “ $g(a)$ ”, case (ii) (c) applies.

For the second argument of D'_{21} , which is “ a ”, case (ii) (a) applies.

$$\begin{aligned}
\bullet D'_{31}\theta_3 &= \neg P(x, g(a))\theta_3 = \neg W'_{13}\sigma\alpha_1 \\
&\quad \neg W'_{13} = \neg P(v, u).
\end{aligned}$$

For the first argument of D'_{31} , which is “ x ”, neither case (i) nor case (ii) applies, since x is a variable.

For the second argument of D'_{31} , which is “ $g(a)$ ”, case (ii) (c) applies.

$$\bullet D'_{32}\theta_3 = \neg D(x, g(a))\theta_3 = \neg W'_{23}\sigma\alpha_1$$

$$\neg W'_{23} = \neg D(v, u).$$

For the first argument of D'_{32} , which is “ x ”, neither case (i) nor case (ii) applies, since x is a variable.

For the second argument of D'_{32} , which is “ $g(a)$ ”, case (ii) (c) applies.

This concludes the case analysis for the formula \mathcal{D} . •

The following theorem shows that by unskolemizing the set of clauses \mathcal{D} mentioned in Lemma 2.3, and Skolemizing any formula F obtained by this unskolemization, we can prove that the case analysis of Lemma 2.3 holds, with the difference that case (ii) (c) will never arise.

Theorem 2.1 Using the same notation as in the previous lemmas, given the set of clauses \mathcal{D} , the unskolemization algorithm can yield a family of formulas \mathcal{K} such that for any $F \in \mathcal{K}$, if we Skolemize F , then for any literal L of $Sk(F)$, where $L = \pm P(d_1, d_2, \dots, d_s)$, say (\pm denotes the sign of L), there exists a literal M of W such that $M = \pm P(b_1, b_2, \dots, b_s)$ (M has the same sign as L) and such that for each i , $1 \leq i \leq s$, the following holds :

- (i) If d_i is a Skolem function (with zero or more arguments), then b_i is a variable which is existentially quantified in W .
- (ii) If d_i is a non-Skolem function (with zero or more arguments), then one of the following holds :
 - (a) b_i is the same function symbol with the same number of arguments, and (i) and (ii) here hold recursively for each corresponding argument of d_i and b_i .
 - (b) b_i is an existentially quantified variable and the function symbol of d_i (with the same arity as d_i) appears in W .

Proof : We prove this theorem by showing that by making some of the nondeterministic choices in the unskolemization algorithm judiciously, a family of formulas \mathcal{K} can be produced by the unskolemization algorithm such that for every $F \in \mathcal{K}$, the statement of the theorem is true.

Using the same notation as before, we have, $\mathcal{D} = D'_1 \wedge D'_2 \wedge \dots \wedge D'_n$. As before, for $1 \leq j \leq n$, we write $D'_j = \{D'_{j1}, D'_{j2}, \dots, D'_{ji}\}$. Now, $\mathcal{D} \wedge Sk(\neg W)$ is unsatisfiable, and in Lemma 2.3, we discussed the relationship between any literal D'_{jk} in \mathcal{D} and the corresponding literal $\neg W'_{k_1 a_1}$ in W which has the property that

$$D'_{jk}\theta_j = \neg W'_{k_1 a_1}\sigma\alpha_1.$$

We also remarked that $W'_{k_1 a_1}\sigma$ is the literal in $Sk(\neg W)$ against which D'_{jk} is resolved in the derivation of the empty clause from $\mathcal{D} \wedge Sk(\neg W)$.

We now show that there is a way of making the nondeterministic choices in Steps 1, 2, and 3 of the unskolemization algorithm such that the properties described in the statement of the theorem will hold. Now, for any clause C in \mathcal{D} , multiple instances of C could be used during the derivation of the empty clause from $\mathcal{D} \wedge Sk(\neg W)$. If k instances of C are used, then we make k copies of the clause C in Step 1. It is clear that if the number of resolutions performed to derive the empty clause from $Sk(H \wedge \neg W)$ is r , then not more than r copies of any one clause are required. After this is done for all clauses of \mathcal{D} , the variables in every clause are renamed so that no two clauses have any variable in common, as specified in Step 1.

Now, for every literal D'_{jk} of every clause of \mathcal{D} we do the following : as in Lemma 2.3, suppose $D'_{jk} = \pm P(d_1, d_2, \dots, d_s)$; we find the corresponding literal $\neg W'_{k_1 a_1}$ in W such that $\neg W'_{k_1 a_1} = \pm P(b_1, b_2, \dots, b_s)$ ($\neg W'_{k_1 a_1}$ has the same sign as D'_{jk}) and $D'_{jk} \theta_j = \neg W'_{k_1 a_1} \sigma \alpha_l$ as we described in Lemma 2.3.

Note that any clause can be used only once for a resolution, since if a clause is needed for k resolutions, then k copies of it were made as described above.

For every i such that $1 \leq i \leq s$, consider d_i and b_i . If either case (i) of Lemma 2.3 holds (i.e. if d_i is a Skolem symbol) or if case (ii)(c) holds, (i.e. if d_i is a non-Skolem function symbol not occurring in W and b_i is a variable), then in Step 2 of the algorithm, we replace the argument d_i by the argument $b_i \leftarrow d_i$ and we say that this argument of D'_{jk} has been **marked**. If d_i is a variable, then we replace the argument d_i by the argument " $b_i \leftrightarrow d_i$ " and call this argument **marked** too. (Note that this symbol " \leftrightarrow " has nothing to do with the implication sign " \rightarrow ".) The marking symbol " \leftrightarrow " has been introduced here to guide Step 3 of the algorithm.

After this process has been completed, we have obtained a new formula MARK $= D''_1 \wedge D''_2 \wedge \dots \wedge D''_n$, where D''_i is the same as D'_i except that zero or more arguments of literals of D'_i are marked in D''_i .

We now describe how Step 3 is performed. Consider in groups all marked arguments of the form " $\alpha_i \leftrightarrow x$ ", each group containing all such marked arguments with the same variable x on the right-hand side of the " \leftrightarrow " sign. Suppose $\alpha_1 \leftrightarrow x$, $\alpha_2 \leftrightarrow x$, ..., $\alpha_n \leftrightarrow x$ are all the marked arguments with x on the right-hand side of the " \leftrightarrow " sign. Consider the set $B = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$. This set contains terms which, during the resolution process, unify with x or with whatever x has been instantiated to so far, and any two terms in this set can be unified with each other. B can contain variables and function/Skolem symbols. Let $B = \text{VAR} \cup \text{FUNC}$, where VAR is the set of variables in B , and FUNC is $B - \text{VAR}$. First we choose one element of VAR, say y_1 (if VAR is non-empty), and replace all the other variables of VAR by y_1 everywhere in the formula MARK. Now consider FUNC. From our

remarks above, since any two elements of FUNC are unifiable, every element of FUNC must be the same function or Skolem symbol f , say, with the same number, say k , of arguments, for some $k \geq 0$. Let

$$ARG_i = \{i^{th} \text{ argument of } z \mid z \in \text{FUNC}\}, \text{ for } 1 \leq i \leq k.$$

Repeat the above process (which was performed for the set B) for each of the k sets $ARG_1, ARG_2, \dots, ARG_k$.

Note that this is not really unification, since variables are not being replaced by the terms with which they unify. We are just unifying all the variables by replacing them by the same variable name.

After this has been done for all the marked arguments of this form, drop the " \rightarrow " signs from the modified formula MARK as well as the elements on the left-hand side of the " \rightarrow " signs. If any two clauses of MARK are now identical, one of them can be dropped. This shows how we can choose which variables to unify in marked arguments in Step 3 of the algorithm.

Now perform Steps 4, 5, 6, 7 of the algorithm, and let the set of formulas obtained be \mathcal{K} . For every F belonging to the set of formulas \mathcal{K} , consider the set of clauses $Sk(F)$. $Sk(F)$ is a set of clauses which is the same as \mathcal{D} , except that :

- (1) Some arguments of literals of clauses of \mathcal{D} have been replaced by Skolem functions (this is true if and only if the corresponding argument in \mathcal{D} was a "marked" function symbol during Step 2), and
- (2) There may be more than one copy of certain clauses of \mathcal{D} (since multiple copies of some clauses of \mathcal{D} were made during Step 1), each of which is possibly altered as mentioned in (1) above.

Now, by Lemma 2.3, for any literal L of \mathcal{D} , where $L = \pm P(d_1, d_2, \dots, d_s)$, say (\pm denotes the sign of L), there exists a literal M of W such that $M = \pm P(b_1, b_2, \dots, b_s)$ (M has the same sign as L) and such that for each i , $1 \leq i \leq s$, the following holds :

- (i) If d_i is a Skolem function (with zero or more arguments), then b_i is a variable which is existentially quantified in W .
- (ii) If d_i is a non-Skolem function (with zero or more arguments), then one of the following holds :
 - (a) b_i is the same function symbol with the same number of arguments, and (i) and (ii) here hold recursively for each corresponding argument of d_i and b_i .
 - (b) b_i is an existentially quantified variable and the function symbol of d_i (with the same arity as d_i) appears in W .
 - (c) b_i is an existentially quantified variable and the function symbol of d_i does not appear anywhere in W .

But note that during the marking process described for Steps 1 and 2, we

marked for unskolemization all arguments which fall under category (ii)(c) above. Therefore in F , all such arguments became existentially quantified variables; and therefore in $Sk(F)$, these variables became new Skolem functions, which fall under category (i) above. Hence in $Sk(F)$, no argument d_i can belong to category (ii)(c), since all arguments of \mathcal{D} falling in category (ii)(c) were unskolemized. Hence all arguments of $Sk(F)$ belong to categories (i), (ii)(a) or (ii)(b) of Lemma 2.3, and our theorem is proved. •

Motivated by the result of Theorem 2.1, we now introduce two definitions.

Definition. A formula F is *more general* than a formula W if the following conditions are satisfied. Suppose we write F and W in prenex-conjunctive normal form so that

$$F = Q^F ((F_{11} \vee F_{12} \vee \dots \vee F_{1i_1}) \wedge (F_{21} \vee F_{22} \vee \dots \vee F_{2i_2}) \wedge \dots \wedge (F_{n1} \vee F_{n2} \vee \dots \vee F_{ni_n}))$$

and

$$W = Q^W ((W_{11} \vee W_{12} \vee \dots \vee W_{1u_1}) \wedge (W_{21} \vee W_{22} \vee \dots \vee W_{2u_2}) \wedge \dots \wedge (W_{m1} \vee W_{m2} \vee \dots \vee W_{mu_m})),$$

where Q^F, Q^W are the quantifier strings of the formulas F and W respectively.

Then for every disjunction $(F_{p1} \vee F_{p2} \vee \dots \vee F_{pi_p})$ of F , where $1 \leq p \leq n$, there is a set of literals $\{W_{j_1 k_1}, W_{j_2 k_2}, \dots, W_{j_i k_i}\}$ of W , where $1 \leq j_i \leq m$, $1 \leq k_i \leq u_{j_i}$, such that given $1 \leq r \leq i_p$, there exists $1 \leq s \leq l$ such that the following relationship holds between F_{pr} and $W_{j_s k_s}$:

Suppose $F_{pr} = \pm P(a_1, a_2, \dots, a_t)$, where \pm denotes the sign of F_{pr} . Then $W_{j_s k_s} = \pm P(b_1, b_2, \dots, b_t)$ ($W_{j_s k_s}$ has the same sign as F_{pr}), and for every k such that $1 \leq k \leq t$,

- (i) If a_k is an existentially quantified variable, then so is b_k .
- (ii) If a_k is a function symbol with u arguments e_1, e_2, \dots, e_u , then either
 - (a) b_k is the same function symbol with the same number of arguments, say f_1, f_2, \dots, f_u , and conditions (i) and (ii) hold for every pair of arguments e_i and f_i , $1 \leq i \leq u$, or
 - (b) b_k is an existentially quantified variable and a_k is a function symbol which occurs in W .

The above definition may seem confusing at first; the following discussion may help to explain intuitively why the term “more general than” has been defined in this way. As we shall see in Theorem 2.5, the number of formulas more general than a given formula is finite under certain elementary syntactic constraints. The reason we want this to be true is the following. Recall that the problem being solved is that we are given a formula H which implies some (unknown) formula W , and we are trying to derive a logical consequence F of H such that

$$H \rightarrow F \rightarrow W.$$

Now, some additional constraint must be placed on F , since otherwise we could simply take $F = H$. We want F to be "close" to W , in some sense. One way to ensure this is to define some constraint on F so that only a finite number of formulas satisfy this constraint. Thus only a finite number of formulas F satisfying this constraint and such that $H \rightarrow F \rightarrow W$ can be derived. This is the reason for defining the "more general than" term above.

The definition itself can be explained by the following simple example. Suppose we have $H_i = P(a_i)$ and $W = \exists xP(x)$. Clearly $H_i \rightarrow W$ for all values of i . Thus there are an infinite number of H_i 's which imply W . Now suppose we require that any function symbol appearing in F , where F satisfies $H \rightarrow F \rightarrow W$, also appear in W . Then clearly we cannot have $F = H_i$ for any i (since none of the a_i 's appear in W); also, there will exist only a finite number of such F 's if we do not allow unnecessary redundancy in F (the exact nature of this redundancy is stated in Theorem 2.5). The way F will be obtained from H_i , then, is by unskolemizing the (non-Skolem) function symbol a_i , and replacing it by an existentially quantified variable, yielding $F = \exists xP(x)$. The definition of "more general than" given above does not allow function symbols which do not appear in W to appear in F if F is more general than W .

Definition. Let F, W be two first-order formulas. We say that $F \preceq W$ if and only if

- (i) F is more general than W
- (ii) $F \rightarrow W$.

Corollary to Theorem 2.1 For every $F \in \mathcal{K}$, F is more general than W .

Proof: Let us write F and W in conjunctive normal form as

$$F = Q^F ((F_{11} \vee F_{12} \vee \dots \vee F_{1i_1}) \wedge (F_{21} \vee F_{22} \vee \dots \vee F_{2i_2}) \wedge \dots \wedge (F_{n1} \vee F_{n2} \vee \dots \vee F_{ni_n}))$$

and

$$W = Q^W ((W_{11} \vee W_{12} \vee \dots \vee W_{1j_1}) \wedge (W_{21} \vee W_{22} \vee \dots \vee W_{2j_2}) \wedge \dots \wedge (W_{m1} \vee W_{m2} \vee \dots \vee W_{mj_m})),$$

where Q^F, Q^W are the quantifier strings of the formulas F and W respectively.

Let ρ be a substitution which replaces existentially quantified variables of F by Skolem functions to obtain $Sk(F)$. Then $Sk(F)$ consists of the following set of clauses :

$$Sk(F) = \{ \{F_{11}\rho, F_{12}\rho, \dots, F_{1i_1}\rho\}, \{F_{21}\rho, F_{22}\rho, \dots, F_{2i_2}\rho\}, \dots, \{F_{n1}\rho, F_{n2}\rho, \dots, F_{ni_n}\rho\} \}.$$

Consider any disjunction $F_{p1} \vee F_{p2} \vee \dots \vee F_{pi_p}$ of F , and consider the corresponding clause $\{F_{p1}\rho, F_{p2}\rho, \dots, F_{pi_p}\rho\}$ of $Sk(F)$. Take any literal $F_{pr}\rho = \pm P(d_1, d_2, \dots, d_t)$, say, of this clause (\pm denotes the sign of $F_{pr}\rho$); then by Theorem

2.1, there exists a literal M of W such that $M = \pm P(b_1, b_2, \dots, b_t)$ (M has the same sign as $F_{pr\rho}$) and such that for all i , $1 \leq i \leq t$, the following holds :

- (i) If d_i is a Skolem function (with zero or more arguments), then b_i is a variable which is existentially quantified in W .
- (ii) If d_i is a non-Skolem function (with zero or more arguments), then one of the following holds :
 - (a) b_i is the same function symbol with the same number of arguments, and (i) and (ii) here hold recursively for each corresponding argument of d_i and b_i .
 - (b) b_i is an existentially quantified variable and the function symbol of d_i (with the same arity as d_i) appears in W .

Now suppose $F_{pr} = \pm P(a_1, a_2, \dots, a_t)$ (F_{pr} obviously has the same sign as $F_{pr\rho}$). If case (i) above holds for d_i , then a_i must be an existentially quantified variable in F .

Since the above is true for every literal of the disjunction $F_{p1} \vee F_{p2} \vee \dots \vee F_{pi_p}$ of F , we can obtain a set S of literals of W such that conditions (i) and (ii) in the definition of "more general than" hold for corresponding literals of $(F_{p1} \vee F_{p2} \vee \dots \vee F_{pi_p})$ and S ; thus by definition, F is more general than W . •

Example 2.3 We continue working where we had left off in Example 2.2. The formula \mathcal{D} which we obtained will now be unskolemized according to the algorithm just presented.

Let us reiterate that the formula W is not normally available to us when we are unskolemizing \mathcal{D} . The choices which we make here based on properties of W will have to be made nondeterministically by the algorithm. We are using W here to show that there exist choices which will result in the derivation of a formula F with the desired properties.

We list the clauses of \mathcal{D} below. Note that two of the clauses of \mathcal{D} are repeated twice. This is because each of these clauses is required to be used twice during the resolution which follows.

1. $\{L(b, g(a))\}$
2. $\{L(b, g(a))\}$
3. $\{L(g(a), a)\}$
4. $\{L(g(a), a)\}$
5. $\{\neg P(x, g(a)), \neg D(x, g(a))\}$.

The clauses of $\text{Sk}(\neg W)$ are given below. The variables have been renamed so that the two clauses do not share variables.

6. $\{\neg L(b, u), \neg L(u, a), P(f(u), u)\}$

7. $\{\neg L(b, w), \neg L(w, a), D(f(w), w)\}$
8. $\{\neg L(b, z), \neg L(z, a), \neg M(a)\}$

The derivation of the empty clause from $\mathcal{D} \wedge \text{Sk}(\neg W)$ proceeds as follows :

- | | | |
|-----|---|-------------------------|
| 9. | $\{\neg L(g(a), a), D(f(g(a)), g(a))\}$ | from clauses 1 and 7 |
| 10. | $\{\neg L(g(a), a), P(f(g(a)), g(a))\}$ | from clauses 2 and 6 |
| 11. | $\{D(f(g(a)), g(a))\}$ | from clauses 3 and 9 |
| 12. | $\{P(f(g(a)), g(a))\}$ | from clauses 4 and 10 |
| 13. | $\{\neg P(f(g(a)), g(a))\}$ | from clauses 5 and 11 |
| 14. | $\{\}$ | from clauses 12 and 13. |

From the above resolutions, following the marking method sketched in the proof of Theorem 2.1, we can write the marked formula MARK as consisting of the following five clauses (refer to the case analysis performed in Example 2.2) :

- 1'. $\{L(b, w \leftarrow g(a))\}$
- 2'. $\{L(b, u \leftarrow g(a))\}$
- 3'. $\{L(w \leftarrow g(a), a)\}$
- 4'. $\{L(u \leftarrow g(a), a)\}$
- 5'. $\{\neg P(f(u) \leftrightarrow x, u \leftarrow g(a)), \neg D(f(w) \leftrightarrow x, w \leftarrow g(a))\}$.

We now perform the unskolemization algorithm step by step.

INPUT : The set of three clauses listed below :

1. $\{L(b, g(a))\}$
2. $\{L(g(a), a)\}$
3. $\{\neg P(x, g(a)), \neg D(x, g(a))\}$.

Step 1 : Make two copies of the clause $\{L(b, g(a))\}$; make two copies of the clause $\{L(g(a), a)\}$; and make one copy of the clause $\{\neg P(x, g(a)), \neg D(x, g(a))\}$. We now have the set of clauses *MULTIPLE_CLAUSES* consisting of the following five clauses :

1. $\{L(b, g(a))\}$
2. $\{L(b, g(a))\}$
3. $\{L(g(a), a)\}$
4. $\{L(g(a), a)\}$
5. $\{\neg P(x, g(a)), \neg D(x, g(a))\}$.

Step 2 : Mark certain arguments of *MULTIPLE_CLAUSES* as follows and get a new set of clauses *MARK* :

- 1'. $\{L(b, w_1 \leftarrow g(a))\}$

- 2'. $\{L(b, u_1 \leftarrow g(a))\}$
- 3'. $\{L(w_2 \leftarrow g(a), a)\}$
- 4'. $\{L(u_2 \leftarrow g(a), a)\}$
- 5'. $\{\neg P(x, u_3 \leftarrow g(a)), \neg D(x, w_3 \leftarrow g(a))\}$.

Step 3 : Replace the six variables u_1, u_2, u_3, w_1, w_2 and w_3 by the new variable Z . MARK now consists of the five clauses :

- 1'. $\{L(b, Z \leftarrow g(a))\}$
- 2'. $\{L(b, Z \leftarrow g(a))\}$
- 3'. $\{L(Z \leftarrow g(a), a)\}$
- 4'. $\{L(Z \leftarrow g(a), a)\}$
- 5'. $\{\neg P(x, Z \leftarrow g(a)), \neg D(x, Z \leftarrow g(a))\}$.

Since the clauses 1' and 2' are identical, and so are the clauses 3' and 4', we can drop clauses 2' and 4'. MARK now consists of the three clauses :

- 1'. $\{L(b, Z \leftarrow g(a))\}$
- 3'. $\{L(Z \leftarrow g(a), a)\}$
- 5'. $\{\neg P(x, Z \leftarrow g(a)), \neg D(x, Z \leftarrow g(a))\}$.

Step 4 : The function "g" in MARK has one argument "a" and no variable arguments, therefore we drop the argument "a" and write MARK as :

- 1'. $\{L(b, Z \leftarrow g)\}$
- 3'. $\{L(Z \leftarrow g, a)\}$
- 5'. $\{\neg P(x, Z \leftarrow g), \neg D(x, Z \leftarrow g)\}$.

Step 5 : We add a universal quantifier for the variable x and get

$$\text{MARK} = \forall x (L(b, Z \leftarrow g) \wedge L(Z \leftarrow g, a) \wedge (\neg P(x, Z \leftarrow g) \vee \neg D(x, Z \leftarrow g))).$$

Step 6 : We have to replace the marked argument " $Z \leftarrow g$ " by a new existentially quantified variable, say y . The function g has zero arguments, therefore we add y to the set \mathbf{A} . We get,

$$\text{MARK} = \forall x (L(b, y) \wedge L(y, a) \wedge (\neg P(x, y) \vee \neg D(x, y))).$$

$$\text{and } \mathbf{A} = \{y\}.$$

Step 7 : We complete the quantifier string of the formula. Since $\mathbf{C} = \emptyset$ and $\mathbf{A} = \{y\}$, there is only one way of completing the quantifier string of the formula. We call the resulting unskolemized formula \mathbf{F} ;

$$\mathbf{F} = \exists y \forall x (L(b, y) \wedge L(y, a) \wedge (\neg P(x, y) \vee \neg D(x, y))).$$

It can easily be verified that \mathbf{F} is more general than W and that $H \rightarrow \mathbf{F} \rightarrow W$, hence $\mathbf{F} \preceq W$. •

Discussion. Recall that for the above example,

$H = \forall x \forall y \forall z \forall w ((Q(y) \vee L(b, y)) \wedge \neg Q(g(a)) \wedge L(b, g(a)) \wedge (R(x, g(a)) \vee \neg P(x, g(a))) \wedge (\neg R(w, z) \vee \neg D(w, z))),$ and

$$W = \exists u \forall v (L(b, u) \wedge L(u, a) \wedge (\neg P(v, u) \vee \neg D(v, u) \vee M(a)))$$

and we obtained $F = \exists y \forall x (L(b, y) \wedge L(y, a) \wedge (\neg P(x, y) \vee \neg D(x, y)))$ by resolution from $Sk(H)$ and then unskolemization. W and F are almost identical (up to variants) except that the last disjunction of W has one more literal (viz. $M(a)$). It is not possible to get a formula F by resolution and unskolemization from $Sk(H)$ which is identical to W up to variants without using tautologies. This is because the predicate symbol “ M ” does not even occur in H , and therefore to introduce it into a clause derived from $Sk(H)$ by resolution, the use of a tautology containing the predicate symbol “ M ” and its negation would be required.

Also note that without the use of our unskolemization algorithm, it would not have been possible to introduce the existential quantifier “ $\exists y$ ” above in F .

Thus we see that it is not always possible to obtain W from H by resolution without the use of tautologies and without unskolemization. Since the use of tautologies in resolution is undesirable (due to the tremendous increase in search space which it would entail), we do not try to derive W from H , but instead settle for a formula F derived from H by resolution and unskolemization, which has the property that

$$F \preceq W.$$

The theorems in this section serve to show that our unskolemization can indeed produce such a formula.

Theorem 2.2 For every $F \in \mathcal{L}$, $\mathcal{D} \rightarrow F$, where free variables in \mathcal{D} are regarded as universally quantified and where \mathcal{L} is the family of formulas obtained by unskolemizing the set of clauses \mathcal{D} according to the unskolemization algorithm.

Proof : Let M be a model for \mathcal{D} with domain D (regarding free variables as universally quantified in \mathcal{D}), and let $F \in \mathcal{L}$. We show that M is also a model for F .

Now, \mathcal{D} and F differ in that all Skolem functions which are arguments of predicates in \mathcal{D} are replaced by existentially quantified variables in F , and in that some functions which are arguments of predicates in \mathcal{D} and which are marked during the marking process of Step 2 are replaced by existentially quantified variables in F . Also, F may contain several copies of some clauses of \mathcal{D} . Suppose $f(\nu_1, \nu_2, \dots, \nu_m)$ is a function in \mathcal{D} which is marked as “ $z \leftarrow f(\nu_1, \nu_2, \dots, \nu_m)$ ” in Step 2 of the algorithm and is replaced by the existentially quantified variable z in F , and suppose x_1, x_2, \dots, x_n are all the (distinct) variables which occur in $\nu_1, \nu_2, \dots, \nu_m$. Then, by the unskolemization process we used, “ $\exists z$ ” comes after “ $\forall x_1$ ”, “ $\forall x_2$ ”, ..., “ $\forall x_n$ ” in the quantifier string of F .

Now, the model M assigns an element d of the domain D of M to the function $f(\nu_1, \nu_2, \dots, \nu_m)$. This element d depends on the mapping assigned to f in M , and on the values of the arguments $\nu_1, \nu_2, \dots, \nu_m$, which in turn depend on the variables x_1, x_2, \dots, x_n and on the constant and function symbols occurring in $\nu_1, \nu_2, \dots, \nu_m$. Thus, given the values for variables x_1, x_2, \dots, x_n , there exists an element d of the domain D such that when d is used in place of $f(\nu_1, \nu_2, \dots, \nu_m)$ in the formula \mathcal{D} , the formula \mathcal{D} is true. But then this means that if we do the above for all such functions in \mathcal{D} which are replaced by existentially quantified variables in F , these elements “ d ” can be used in place of the corresponding existentially quantified variables “ z ” in F and will result in the formula F being true under interpretation M (since each such existentially quantified variable z depends on the universally quantified variables x_1, x_2, \dots, x_n in F , and possibly some others). Hence M is also a model for F , and therefore $\mathcal{D} \rightarrow F$. •

Corollary to Theorem 2.2 For every $F \in \mathcal{L}$, $H \rightarrow F$.

Proof: Recall that \mathcal{D} is a set of clauses derived by resolution from $Sk(H)$; therefore

$$Sk(H) \rightarrow \mathcal{D} \text{ (where free variables are regarded as universally quantified),}$$

hence

$$Sk(H) \rightarrow F \text{ (since } \mathcal{D} \rightarrow F \text{ from Theorem 2.2),}$$

hence

$$H \rightarrow unsk(F) = F; \text{ and so}$$

$$H \rightarrow F.$$

This is true for any $F \in \mathcal{L}$, and therefore the corollary is proved. •

Theorem 2.3 For any formula A , there is a way of marking $Sk(A)$ so that the formula produced by the unskolemization algorithm will be A .

Proof: Consider the set of clauses $Sk(A)$. We mark the terms of $Sk(A)$ as follows : replace all occurrences of a Skolem function f by “ $x_f \leftarrow f$ ”, x_f being a new variable not occurring elsewhere in $Sk(A)$. This is done for all Skolem functions f . Then unskolemize this marked set of clauses using the unskolemization algorithm. The algorithm will produce A (or a formula equivalent to A) as its only output, since there is only one optimal way of ordering the quantifiers of the resulting formula. •

The following theorem shows that using our unskolemization algorithm as described, the algorithm will produce at least one formula F such that $F \rightarrow W$.

Theorem 2.4 There exists $F \in \mathcal{K}$ such that $F \rightarrow W$, where \mathcal{K} is the family of formulas defined in Theorem 2.1.

Proof: Let $F \in \mathcal{K}$; we write F in prenex-conjunctive normal form so that

$$F = Q^F (A_1^u \wedge A_2^u \wedge \dots \wedge A_m^u) \text{ (here } Q^F \text{ is the quantifier string of } F).$$

Recall that

$$\mathcal{D} = D'_1 \wedge D'_2 \wedge \dots \wedge D'_n,$$

where

$$D'_j = \{D'_{j1}, D'_{j2}, \dots, D'_{ji_j}\} \text{ for each } j \text{ such that } 1 \leq j \leq n.$$

Let $A_j^u = A_{j1}^u \vee A_{j2}^u \vee \dots \vee A_{ji_j}^u$ for each j such that $1 \leq j \leq m$.

Let ρ be the substitution which replaces existentially quantified variables of F by Skolem functions to get $Sk(F)$; i.e. $F\rho = Sk(F)$, and letting

$Sk(F) = ((A_{11} \vee A_{12} \vee \dots \vee A_{1i_1}) \wedge (A_{21} \vee A_{22} \vee \dots \vee A_{2i_2}) \wedge \dots \wedge (A_{m1} \vee A_{m2} \vee \dots \vee A_{mi_m}))$, we have

$$A_{jk} = A_{jk}^u \rho \text{ for all } j, k \text{ such that } 1 \leq j \leq m, 1 \leq k \leq i_j.$$

Now, $\mathcal{D} \wedge T$ is unsatisfiable (recall that $T = Sk(\neg W)$), and in Lemma 2.1 we found a set of ground clauses T' such that $\mathcal{D} \wedge T'$ is unsatisfiable. Then we looked at every literal D'_{jk} in \mathcal{D} and found the corresponding literal, say L , in $Sk(\neg W)$ against which D'_{jk} was resolved during the derivation of the empty clause from $\mathcal{D} \wedge T$ (see proof of Lemma 2.3). Then for every argument which was a function symbol in the literal D'_{jk} and did not occur in W , and which was unified with a variable in the literal L , we "marked" this function argument, and unskolemized it so that every formula F in \mathcal{K} (the family of unskolemized formulas resulting from \mathcal{D}) had an existentially quantified variable in that position (see proof of Theorem 2.1).

For any $F \in \mathcal{K}$, \mathcal{D} and F are formulas which are identical in structure; the only difference is that some functions and all Skolem functions of \mathcal{D} are replaced by existentially quantified variables in F , and that F may contain several copies of some clauses of \mathcal{D} . Two functions were replaced by the same existentially quantified variable if and only if the two functions unified with the same variable during the derivation of the empty clause from $\mathcal{D} \wedge Sk(\neg W)$, or if the functions unified with two variables which unified with each other during the course of the derivation of the empty clause from $\mathcal{D} \wedge T$.

Therefore the only difference between \mathcal{D} and $Sk(F)$ is that all Skolem functions of \mathcal{D} are replaced by Skolem functions in $Sk(F)$ with possibly different arguments; and those functions of \mathcal{D} which are resolved against variables in literals of $Sk(\neg W)$ and which do not occur in W are replaced by Skolem functions in $Sk(F)$. Since any n functions in \mathcal{D} which resolved against the same variable in T , or which resolved against some variables in T which unified with each other during the course of the resolution, were replaced by the same existentially quantified variable in F , the Skolem function replacing that variable in $Sk(F)$ will be the same for all n of these argument positions, and therefore they can all still be resolved against the same variables in T against which they were resolved during the course of the derivation

of the empty clause from $\mathcal{D} \wedge T$.

So all we need to do here is to show that there exists some $F \in \mathcal{K}$ such that the empty clause can be derived from $Sk(F) \wedge T$ by using exactly the same sequence of resolutions which was used to derive the empty clause from $\mathcal{D} \wedge T$. We do this by showing that there is a certain order in which we can process the set of constraints \mathbf{C} for \mathcal{D} (which is done in Step 7 of the unskolemization algorithm) which will give the resulting formula F the property that $Sk(F) \wedge T$ is unsatisfiable.

In Lemma 2.3, for every literal D'_{jk} of \mathcal{D} we had found a literal $\neg W'_{k_1 a_1}$ of W such that

$$D'_{jk} \theta_j = \neg W'_{k_1 a_1} \sigma \alpha_l.$$

Let the corresponding literal in $Sk(F)$ be A_{jk} .

Let $D'_{jk} = P(d_1, d_2, \dots, d_s)$,

$$\neg W'_{k_1 a_1} = P(b_1, b_2, \dots, b_s),$$

$A_{jk} = P(a_1, a_2, \dots, a_s)$ (without loss of generality we have assumed that all three literals here are positive; the same result can easily be seen to hold if all three literals are negative).

Consider any constraint (y, z) in \mathbf{C} , where z is a variable in F which was replaced by a Skolem function, say a_i , in A_{jk} . We will show that this constraint must also hold in W for the arguments with which y and z unify in $Sk(\neg W)$, if these arguments are universally and existentially quantified respectively in W ; in other words, the existential quantifier for the variable in W unifying with z must come after the universal quantifier for the variable in W unifying with y . Now, a_i is a function containing y as an argument, say $a_i = g(y, \text{other arguments})$. Either y appears elsewhere in A_{jk} , or it doesn't. If it doesn't, then we don't need to worry about the constraint (y, z) since it is not relevant for this particular literal. If it does, then suppose a_j contains y . Now consider b_i and b_j . Since a_i is a Skolem function, b_i must be an existentially quantified variable, say $b_i = v$, in W . Since a_j contains y , b_j contains a term, say u , which unifies with y ; u could either be a universally quantified variable, an existentially quantified variable, or a function symbol. If one of the latter two is true, we need not worry about it; if the first of these is true, i.e. if u is a universally quantified variable, then we must show that the constraint that $\forall u$ must precede $\exists v$ in the quantifier string of W holds for W .

Suppose it doesn't. Then in $\neg W$, " $\exists u$ " comes after " $\forall v$ " in the quantifier string for $\neg W$. Therefore σ assigns a Skolem function, say β , to u which contains v as an argument (recall that $\neg W \sigma = Sk(\neg W)$); say the assignment is : $u \leftarrow \beta(v, \text{other arguments})$. The substitution σ leaves v unchanged, since v is universally quantified in $\neg W$.

$$\text{Now, } D'_{jk} \theta_j = \neg W_{j_1 k_1} \sigma \alpha_l.$$

Therefore $y\theta_j = \beta(v, \text{other arguments}) \alpha_l$
and $g(y, \text{other arguments}) \theta_j = v\alpha_l$.

But this means that y unifies with $\beta(v, \text{other arguments})$ and v unifies with $g(y, \text{other arguments})$. From this we see that y gets unified with a term containing y , which is a contradiction since such a unification cannot succeed due to occur check.

Hence our assumption must be wrong, i.e. the quantifier “ $\forall u$ ” must precede “ $\exists v$ ” in the quantifier string for W .

We have shown that any constraint (y, z) in \mathbf{C} must hold for the corresponding arguments in W , if the arguments u and v (say) corresponding to y and z are universally and existentially quantified respectively.

Now, we have the freedom to process the members of the constraints set \mathbf{C} in any order we wish. Process them in the order which will make the order and position of the newly inserted existential quantifiers in the quantifier string of F the same as in the quantifier string of W , relative to the universal quantifiers which are already present in the partially completed quantifier string of F . Name the formula constructed in this way “ \mathbf{F} ”; then $\mathbf{F} \rightarrow W$. •

Corollary to Theorem 2.4: There exists $\mathbf{F} \in \mathcal{K}$ such that $\mathbf{F} \preceq W$.

Proof : From the Corollary to Theorem 2.1, Theorem 2.4, and the definition of \preceq . •

Theorem 2.5 $\{F \mid F \preceq W\}$ is finite up to variants, assuming that if F is written in conjunctive normal form, then no two disjunctions of F are identical, and no disjunction of F contains more than one occurrence of the same literal.

Proof : We show that the set $\{F \mid F \text{ is more general than } W\}$ is finite up to variants subject to the above condition, namely that if F is written in conjunctive normal form, then no two disjunctions of F are identical, and no disjunction of F contains more than one occurrence of the same literal. Suppose a formula W is given, and suppose F is a formula which is more general than W . Let both W and F be given in prenex-conjunctive normal form as :

$$F = Q^F ((F_{11} \vee F_{12} \vee \dots \vee F_{1i_1}) \wedge (F_{21} \vee F_{22} \vee \dots \vee F_{2i_2}) \wedge \dots \wedge (F_{n1} \vee F_{n2} \vee \dots \vee F_{ni_n}))$$

and

$$W = Q^W ((W_{11} \vee W_{12} \vee \dots \vee W_{1j_1}) \wedge (W_{21} \vee W_{22} \vee \dots \vee W_{2j_2}) \wedge \dots \wedge (W_{m1} \vee W_{m2} \vee \dots \vee W_{mj_m}))$$

where Q^F, Q^W are the quantifier strings of the formulas F and W respectively. Since F is more general than W , by definition for every disjunction $(F_{p1} \vee F_{p2} \vee \dots \vee F_{pi_p})$ of F , there is a set of literals $\{W_{j_1 k_1}, W_{j_2 k_2}, \dots, W_{j_l k_l}\}$ of W such that given $1 \leq r \leq i_p$, there exists $1 \leq s \leq l$ such that the following relationship holds between F_{pr} and $W_{j_s k_s}$:

Suppose $F_{pr} = \pm P(a_1, a_2, \dots, a_t)$, where \pm denotes the sign of F_{pr} . Then $W_{j_s k_s} = \pm P(b_1, b_2, \dots, b_t)$ ($W_{j_s k_s}$ has the same sign as F_{pr}), and for every k such that $1 \leq k \leq t$,

(i) If b_k is an existentially quantified variable, then a_k is either an existentially quantified variable, a universally quantified variable, or a function symbol which occurs in W .

(ii) If b_k is a universally quantified variable, then so is a_k .

(iii) If b_k is a function symbol with u arguments e_1, e_2, \dots, e_u , then a_k is either :

(a) the same function symbol with the same number of arguments, say f_1, f_2, \dots, f_u , and conditions (i), (ii) and (iii) hold for every pair of arguments e_i and f_i , $1 \leq i \leq u$, or

(b) a_k is a universally quantified variable.

From the above analysis, it can be seen that only a finite number of distinct literals F_{pr} (up to variants) can be constructed which satisfy these conditions. But then there exist only a finite number of formulas F made up of conjunctions of disjunctions of such literals, provided no two such disjunctions are identical, and no disjunction of F contains more than one occurrence of the same literal.

Hence the number of formulas which are more general than W is finite up to variants, subject to the conditions in the statement of the theorem; this means that $\{F \mid F \preceq W\}$ is also finite up to variants subject to the same conditions. •

Theorem 2.6 If F_1, F_2 and W are three formulas such that

$$F_1 \preceq W, F_2 \preceq W,$$

then

$$(F_1 \wedge F_2) \preceq W, (F_1 \vee F_2) \preceq W.$$

Proof : Since $F_1 \preceq W, F_2 \preceq W$, therefore we know that

$$F_1 \rightarrow W, F_2 \rightarrow W$$

and therefore

$$(F_1 \wedge F_2) \rightarrow W, (F_1 \vee F_2) \rightarrow W.$$

Also, since each of F_1 and F_2 are more general than W , from the definition of "more general than" it can be seen that both $F_1 \wedge F_2$ and $F_1 \vee F_2$ are more general than W . Hence

$$(F_1 \wedge F_2) \preceq W, (F_1 \vee F_2) \preceq W,$$

by definition. •

2.4 Summary

We have seen that given formulas H and W such that $H \rightarrow W$, we can derive a Skolemized formula \mathcal{D} by resolution from $Sk(H)$ which can be unskolemized in such a way that the resulting formula \mathbf{F} has the following properties :

- (i) $H \rightarrow \mathbf{F} \rightarrow W$
- (ii) $\mathbf{F} \preceq W$.

We also saw that $\{F \mid F \preceq W\}$ is finite up to variants, subject to the restriction that if F is expressed in conjunctive normal form, then no two disjunctions of F are identical, and no disjunction of F contains more than one occurrence of the same literal. Note that this method of deriving \mathbf{F} does not require using tautologies during the resolution process.

However, it will happen that we know H and do not know W . In such a case, we will have to derive all possible formulas \mathcal{D} by resolution from $Sk(H)$, mark \mathcal{D} in all possible ways, and apply the unskolemization algorithm to all such marked formulas. The unskolemization algorithm will produce a family of formulas \mathcal{K} , out of which (at least) one formula \mathbf{F} will have the properties described above.

3. Mechanical generation of loop invariants for program verification

In this chapter, we will develop a method for mechanically deriving loop invariants for a flowchart program. No complete method can exist for automatically deriving loop invariants for all possible programs, since by Cook's completeness result [Cook 78], there exist program loops for which no suitable loop invariants exist, unless the language being used is "expressive" in some sense (see [Loeckx and Sieber 87] for a detailed coverage of this topic). Any calculus based on attaching first-order formulas to arcs of flowchart programs may be incomplete because the set of possible values on an arc of the flowchart may not be first-order definable [Wand 78]. Most of the attempts made at developing methods for automatically generating loop invariants have been in the nature of heuristics so far, and thus none of these methods has been complete in any sense. In contrast, we can make the following completeness claim about our method : given any program loop, if a loop invariant exists for that loop in a given first-order language relative to a given finite set of first-order axioms, then our method can produce a valid loop invariant for that loop. Of course, not all theories of interest can be expressed by a finite collection of first-order axioms.

In what follows, we first describe in detail past work in the area of program verification. We then explain how to apply the theory developed in Chapter 2 for mechanically generating loop invariants.

3.1 Past work

In an age where more and more reliance is being placed upon computer software in all spheres of life, there is bound to be some concern about the correctness of programs being written and used. According to Elspas et al. [Elspas et al. 72], when we compare programs written in the 1970s to those written in the 1960s, the number of errors per line of debugged code is undoubtedly lower than before; however, since the size of programs written in the 1970s is much larger than in 1960, the number of errors per program is more or less unchanged. The traditional

manner of assuring program correctness is to run a program on “representative” data sets and verify that the results obtained are indeed what is expected. This method, however, cannot provide anything more than some degree of confidence that the program will always fulfill its objective, and is by no means a guarantee of the program’s correctness. Moreover, with this method, design flaws often are detected only after a large investment has been made to develop the system to a point where it can be run. The rebuilding that is caused by the late detection of these flaws contributes significantly to the high cost of software construction and maintenance [Good 85]. For this reason, computer scientists started turning their attention to formal mathematical methods for rigorously proving program correctness. The inductive assertions method for program verification was developed by Floyd in 1967 [Floyd 67] and is now the basis for a large number of automated program verification systems. This method requires that the user annotate the loops of the program with *inductive assertions* (also called *loop invariants*) which are invariants of the loops. However, specifying inductive assertions for program loops is a redundant, tedious and error-prone task for the programmer [German and Wegbreit 75]. Therefore an area of research which is of great interest and potentially of great use to the the community is the automatic derivation of inductive assertions for program loops. This chapter describes an iteration method for automatically deriving loop invariants for flowchart programs.

Dijkstra [Dijkstra 89] made a strong case for the indispensability of formal program verification, arguing that software bugs are programming errors which can be eliminated by formally verifying programs. His article evoked a heated response from many members of the community, some of them arguing that since there is always a human element involved, be it in programming or in program verification, and since human beings are imperfect, error-free programs are virtually an unattainable dream. Others have even questioned the desirability of formal program verification [De Millo et al. 79]. The existence of such diametrically opposed viewpoints indicates that this issue will not resolved in the near future.

The following is a review of some program verification systems which have been built in the past. These can be divided, for our purposes, into two broad categories : those in which the user has to supply the loop invariants for the loops in the program, and those in which the program verifier provides assistance to the user in deriving the loop invariants for the program being verified.

In his pioneer system, King [King 69] describes a program verifier which he wrote and implemented for his Ph.D. dissertation. He regards his system as a first step toward developing a “verifying compiler”, which not only performs the translation of a program to machine executable form, but also attempts to prove

that the program is "correct". The system is written in assembly language running on an IBM 360 and operates on programs written in a simple programming language for integer arithmetic. The model of computation he uses is also described in [King 71]. In this model, programs can consist of three types of statements : assignment statements, tests, and the "halt" statement. The method described is essentially the inductive assertions method of Floyd [Floyd 67]. The inductive assertions for the loops of the program to be verified have to be provided by the programmer. The formal analysis of the program produces verification conditions that must be proved to be theorems over integers. These theorems are proved by powerful formula simplification routines and specialized techniques for integer expressions. King lists nine examples which he used to test his system. The verification of the ninth example was unsuccessful. Deutsch [Deutsch 73] and others later found that the inductive assertion for the loop of this program was not strong enough and that the program could not be proved correct if that inductive assertion was used.

Deutsch describes an interactive program verifier called PIVOT (Programmer's Interactive Verification and Organizational Tool) in his Ph.D. dissertation [Deutsch 73]. This work is very similar to that of King [King 69] in certain areas such as representation and simplification of arithmetic expressions. The verifier is based on Floyd's inductive assertions method. An algebraic, statement-oriented language is used for the programs to be analyzed. Deutsch added an interactive facility to his verifier to cope with the inability of his system to generate inductive proofs automatically; the user can thus guide the proof procedure if necessary.

Another program verification system in which the user provides the inductive assertions for the program being proved correct is one described by Cooper [Cooper 71]. He describes a system aimed at building routines to be used in mechanical and mechanically-aided proofs about the correctness and convergence of programs. His system contains an arithmetic simplifier which can reduce arithmetic expressions to a more standard form and also performs conversion to conjunctive or disjunctive normal form. Another component of the system is an implementation of a modified version of the Presburger arithmetic algorithm (a formula is said to be a formula of Presburger arithmetic if it is formed from algebraic expressions, only allowing variables, constants, addition and subtraction, the arithmetic relations $<$ and $=$, the propositional calculus logical connectives, and universal or existential quantification). Programs are regarded as being made up of blocks, and relations describing the properties of the blocks are attached to the blocks, much in the same fashion as attaching predicates to a point in a program as described by Floyd [Floyd 67]. However, in contrast to the Floyd approach, the equations for a loop block are inherently second-order. These can be changed to first-order equations if all the

loop predicates are specified by the programmer. Cooper goes on to say that iterating a loop a few times soon gives the programmer a good idea of what a suitable invariant might be; however, he has not come up with a program which could do this automatically.

Good et al. [Good et al. 75] report the development of an interactive program verification system for verifying Pascal programs. In their system, the user is primarily responsible for correctness proofs for programs. All loop invariants must be provided by the programmer. The verification condition generator is an implementation of the axioms and rules of inference which constitute the axiomatic definition of Pascal. The theorem prover used is based on natural deduction, which facilitates computer-user interaction, since the theorems being proved are expressed in a form more intuitively comprehensible to the user. The prover is interactive and is based on the premise that if it can construct a proof automatically, it will do so fairly quickly; if a theorem has not been proved within some specified time limit, the prover stops and waits for interactive direction. A sorting program taken from [King 69] is given as an example illustrating the working of the verifier.

Suzuki [Suzuki 75] describes methods for verifying programs written in a subset of Pascal, which may contain data structures such as *array*, *pointer*, and *record*, and control structures such as *while*, *repeat*, *for*, *procedure*, *function*, and *coroutine*. According to Suzuki, the two major hurdles in automatic program verification are the following. First, the language used to express assertions is usually first-order predicate logic, which he claims is unnatural. Secondly, general-purpose theorem provers are usually inadequate for proving the verification conditions generated from a given program. His system allows users to introduce new symbols by documentation in the form of three simple kinds of statements which are used by the prover as rewriting rules to expand new symbols, reduction strategies which state that some expressions are reduced to others under specified conditions, and goal-subgoal strategies which state that certain well-formed formulas are true if certain others are true. The basis of the deduction mechanism used is a Gentzen-type formal system. Suzuki illustrates the working of his verifier by demonstrating correctness proofs for Floyd's Treesort and Hoare's FIND programs. The loop invariants for programs verified by this method have to be supplied by the user.

Polak [Polak 81] describes the design, implementation and verification of a compiler for a Pascal-like language. The Stanford Verifier [Stanford 79] is used to give a complete formal machine-checked verification of the compiler. The author regards the verification as an integral part of program development. The verification system used is based on Hoare's calculus [Hoare 69]. Loop invariants must be supplied by the programmer.

The IOTA project [Nakajima and Yuasa 83] was motivated by a need to develop a mechanizable verification method for programming with modules. This led to the design of a programming and specification language IOTA for modular programming and then to the development of a total programming system. The system provides an integrated environment to enhance the goal of modular programming and consists of five major subsystems : developer, debugger, verifier, prover, and executor. The program verifier is based on Hoare's system and the loop invariants are provided by the programmer.

Good [Good 85] describes the Gypsy verification environment, which is a large, interactive computer program that supports the construction of formal, mathematical proofs about the behavior of software systems. It contains tools for supporting the normal software development process as well as tools for constructing formal proofs. The environment is based on the Gypsy language [Good et al. 78]. The external environment consists of data objects, each of which has a name and value, which are changed as a result of implementing a program. Internal data objects can be created and used by an implementation of a program to accomplish its effect. Gypsy provides a means of stating both internal and external specifications of a program, which define constraints of its implementation. From these specifications, the verification conditions for a program can be built and the program can be verified by Floyd's inductive assertions method. These verification conditions are then proved with an interactive proof checker, which relies heavily on user guidance. Some examples illustrating the use of the system are given. A measure of the efficiency of the system is given for two examples in terms of the number of proved executable Gypsy lines per work-day per CPU-hour.

German and Wegbreit [German and Wegbreit 75] describe a system which provides assistance to the user in synthesizing correct inductive assertions. The system is called VISTA and it uses four principal methods to obtain inductive assertions : 1) symbolic evaluation in a weak interpretation, 2) combining output assertions with loop exit information to obtain trial loop assertions, and generalizing these where necessary, 3) propagating valid assertions forward through the program, modifying them as required by the program transformations, and 4) extracting information from proofs that fail in order to determine how assertions should be strengthened. None of these methods are complete, but when coupled together they can help in automatically deriving inductive assertions in a number of cases. The authors believe that the language for specifying assertions should be improved to facilitate specification of assertions by the programmer. Also, the theorem prover which is used by the program verifier should have the capability of efficiently checking the validity of a formula and a number of slightly varied formulas. They have succeeded

in generating inductive assertions for the first seven examples in [King 69] and in extending the incomplete inductive assertion in [Example 9, King 69] to the complete inductive assertion. The theorem prover used is PIVOT, described in [Deutsch 73].

An interactive approach to program verification for Pascal programs is described by von Henke and Luckham [von Henke and Luckham 75]. They are of the view that a program verifier is a tool which can sometimes enable a programmer to gain a degree of certainty about his or other people's programs. Thus the program verifier aids the user in situations where documentation is incomplete, the program is unfinished or badly written, or the data structures are non-standard. The verification system is the same as that described in [Suzuki 75]. An example is given where a programmer writes a partially incomplete program for performing unification; the program also contains errors. The verification system used interactively participates in locating errors and omissions in the program. The methodology given is not complete, and neither is it intended to be; many of the problems which arise during a verification involve the user in making choices and decisions. This verifier is intended for use in conjunction with other programming facilities.

Some work has been done in the past on the subject of deriving inductive assertions mechanically. Wegbreit [Wegbreit 73] describes heuristic methods for mechanically deriving loop invariants from their boundary conditions and for mechanically completing partially specified loop predicates. The method uses the output predicate to derive suitable loop predicates by dragging the output predicate backwards through the program and modifying it suitably when passing through the statements of the program. Another alternative he gives is to take a programmer-supplied inductive assertion, which contains the "essential idea" of a loop, and mechanically fill in the details to obtain a complete and correct loop predicate. Wegbreit described some domain-dependent and some domain-independent heuristics for deriving loop predicates. He starts by using the weakest possible loop invariant for a particular loop which will satisfy one of the verification conditions, and tries to strengthen this loop invariant using a number of heuristics. The heuristics include strengthening the current loop invariant by dropping some disjuncts, propagating predicates backwards through the program, adding expressions which are equal to zero to one side of an equality, multiplying one side of an equality by an expression which is equal to one, eliminating variables from inequalities using transitivity, etc. These heuristics are illustrated with the help of several examples, all involving simple loop programs or nested loop programs. A short example illustrating this approach is the following: the flowchart of Figure 3.1 computes the quotient Q and the remainder R of integer X divided by integer Y . Here the input predicate ϕ is given by $X \geq 0 \wedge Y > 0$, and the output predicate $\psi = (X = QY + R \wedge 0 \leq R \wedge R < Y)$.

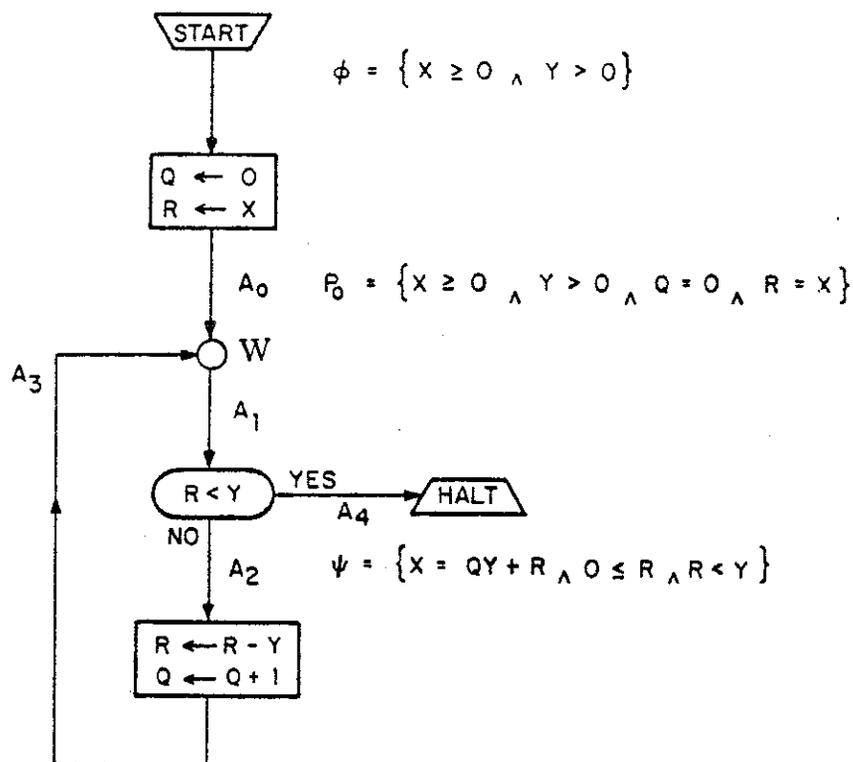


Figure 3.1 Calculating the quotient and remainder of two numbers

This implies that at arc A_0 , the predicate $P_0 = (X \geq 0 \wedge Y > 0 \wedge Q = 0 \wedge R = X)$ holds. To verify the flowchart, it suffices to find a loop predicate P_1 at arc A_1 such that

$$(E1) P_0 \rightarrow P_1$$

(E2) $P_1 \wedge \delta(1, 2, 3, 1) \rightarrow P_1'$, where $\delta(1, 2, 3, 1)$ is the transformation due to the flowchart path A_1, A_2, A_3, A_1 in that order, and P_1' is predicate P_1 with the values of the variables altered by going through path A_1, A_2, A_3, A_1 once.

(E3) $P_1 \wedge \delta(1, 4) \rightarrow \psi'$, where $\delta(1, 4)$ is the transformation due to the flowchart path A_1, A_4 in that order, and ψ' is predicate ψ with the values of the variables altered by going through path A_1, A_4 once.

The standard means for generating a loop predicate is to use (E3) and start with trial choice of $P_1 = (\delta(1, 4) \rightarrow \psi')$. Here, this gives $P_1 = (R < Y \rightarrow (X = QY + R \wedge 0 \leq R \wedge R < Y))$. Converting to disjunctive form and simplifying,

$P_1 = (R \geq Y \vee (X = QY + R \wedge 0 \leq R))$. To verify the flowchart, it suffices to prove that with this choice of P_1 , (E1) and (E2) are each valid. (E1) is

$$(X \geq 0 \wedge Y > 0 \wedge Q = 0 \wedge R = X) \rightarrow (R \geq Y \vee (X = QY + R \wedge 0 \leq R))$$

which is valid. However, (E2) is

$$(R \geq Y \vee (X = QY + R \wedge 0 \leq R)) \wedge R \geq Y \wedge R' = R - Y \wedge Q' = Q + 1 \rightarrow (R' \geq Y \vee (X = Q'Y + R' \wedge 0 \leq R'))$$

which, while satisfiable, is not valid. This suggests that the trial choice for P_1 should be replaced by a stronger one. Dropping a disjunct is a possible strengthening transformation; plausibility arguments suggest that the disjunct to drop is the one arising from $\delta(1,4)$. Hence consider the next trial choice $P_1 = (X = QY + R \wedge 0 \leq R)$. (E1) remains valid; (E2) becomes

$$(X = QY + R \wedge 0 \leq R) \wedge R \geq Y \wedge R' = R - Y \wedge Q' = Q + 1 \rightarrow (X = Q'Y + R' \wedge 0 \leq R')$$

which is also valid. Hence, this choice of P_1 is said to validate (E1) and (E2), and the flowchart is verified.

It appears that the method would not be as easy to apply to programs with arbitrary loop structures. Wegbreit mentions that such programs could be handled by obtaining an approximation to one loop predicate by a finite expansion to some depth i , and using this approximation to obtain another loop predicate, and so on. This heuristic is not illustrated in any example. These methods were the result of hand simulations and were not actually implemented. Wegbreit mentions that a breadth-first search capability would be required in the implementation of this system. He believes that this method would be successful when applied to programs which have their loops tagged with assertions of varying degrees of completeness: some complete, some partial, and some untagged.

The efforts of Katz and Manna [Katz and Manna 73] are also directed towards automatically deriving loop invariants. They describe two general approaches for doing so; the first is the top-down approach, in which the loop invariant is obtained by analyzing the predicates which are known to be true at the entrances and exits of the loop, and the second is the bottom-up approach, in which the loop invariant is generated directly from the statements in the loop. The top-down approach is similar to that described by Wegbreit in [Wegbreit 73]. The bottom-up approach tries to find general expressions for the values of the program variables after n loop iterations and then eliminate n from these expressions. As a brief example, suppose that program variables y_1 and y_2 are changed only in the assignments $(y_1, y_2) \leftarrow (y_1 + xy_3, y_2 + 5y_3)$ inside a loop; then

$$y_1^{(n)} = y_1^{(0)} + x \sum_{i=1}^n y_3^{(i-1)}, y_2^{(n)} = y_2^{(0)} + 5 \sum_{i=1}^n y_3^{(i-1)}$$

where $y^{(i)}$ denotes the value of y after the i^{th} loop iteration. Therefore

$$\frac{y_1^{(n)} - y_1^{(0)}}{x} = \sum_{i=1}^n y_3^{(i-1)} = \frac{y_2^{(n)} - y_2^{(0)}}{5}.$$

Assuming we know that the initial values of y_1 and y_2 upon first entering the loop are $y_1^{(0)} = 1$ and $y_2^{(0)} = 0$, we obtain the invariant $5(y_1 - 1) = xy_2$.

All the given heuristics apply to loops without branches. A different set of heuristics is given for programs with arrays; underlying these heuristics is the assumption that arrays are used to treat a large number of variables in a uniform manner and not as a collection of unrelated variables. Assertions about arrays are assumed to be of the form $\forall j [\langle j - index \rangle \rightarrow \langle j - array \rangle]$ or $\exists j [\langle j - index \rangle \wedge \langle j - array \rangle]$, where $\langle j - index \rangle$ is a claim on the indices of the array and $\langle j - array \rangle$ is the claim which is made about the array elements themselves. The rules given in this paper do not comprise a general system for finding inductive assertions; rather, they just provide some useful guidelines which could help in finding some inductive assertions which commonly occur in practice.

Others have done research on methods for automatically deriving inductive assertions for specific types of programs. Caplain [Caplain 75] describes a technique applicable to numerical programs, which is based on expressing the transformation of the n variables in a loop by a $n \times n$ matrix; i.e. if \vec{X} is a vector of n variables, then express the transformation effected upon \vec{X} in the loop by writing $\vec{X}^{transformed} = [A]\vec{X} + \vec{C}$, where $[A]$ is a $n \times n$ matrix and \vec{C} is a constant vector. If $[A]$ is diagonalizable, i.e. if $[A]$ can be expressed as $[A] = [P]^{-1}[D][P]$, where $[D]$ is a diagonal matrix, then a set of "basic invariants" can be exhibited which has the property that every invariant expression can be expressed as a function of the basic invariants, and that no basic invariant can be expressed as a function of the others. Thus this basis of invariant expressions is minimal and sufficient for the purpose of any proof, because it subsumes any other invariant. For non-linear transformations, he suggests finding a change of variables which will linearize the transformation and make it diagonalizable. Various rules are given for dealing with loops with branches. The rules given are not complete and require a rather sophisticated mechanization which would probably succeed only with interactive intervention of the user. The application of a similar approach for non-numerical programs would require an elaborate axiomatization of the domain type, making the outcome of such research uncertain.

Some work has also been done in program verification involving higher order logics. Manna [Manna 70] shows that it is possible to formalize all properties regularly observed in deterministic and non-deterministic algorithms in second-order predicate calculus. He also shows that for any algorithm, it suffices to know how to formalize its partial correctness by a second-order formula in order to formalize all other properties by second-order formulas.

An oft-repeated complaint against general-purpose theorem provers is that they are incapable of efficiently handling the proofs which arise during program verification. In the opinion of Elspas et al. [Elspas et al. 72], special-purpose theorem provers need to be built for program verification, e.g. King's system [King 70] or that of Rulifson et al. [Rulifson et al. 71]. They do not think that it is feasible for a machine to generate loop invariants without human intervention at some stage. Theorem provers have in general been most successful when applied to proofs of theorems in relatively small axiomatic domains like group theory and lattice theory and have been less successful on problems in fields such as number theory. One reason for this is the necessity of including mathematical induction among the axioms of the theory, which cannot be done in first-order logic. A way around this is to introduce an induction axiom for each predicate that might conceivably be needed in a certain proof.

Some special-purpose theorem provers have been built for program verifiers. King [King 70] describes an interpretation-oriented theorem prover over integers built as part of a program verifier (described more fully in [King 69]). The task of the theorem prover is to prove theorems in which the functions are the arithmetic operators $+$, $-$, $*$, div , \uparrow , mod , and abs , and the predicates are $>$, $<$, \geq , \leq , $=$, and \neq . Since these functions and predicates have a fixed interpretation, it is possible to use highly specialized and domain-specific procedures in the theorem prover. Note that in general, it is theoretically impossible to construct a program which can decide the validity of any expression in this general class of expressions (see [Davis et al. 61]). The theorem prover consists of two parts: the formula simplifying system and the linear prover. The formula simplifying system maintains expressions in a certain "normal" form. Several simplifying procedures are applied to eliminate subsumed clauses and to reduce sets of equalities and inequalities into smaller sets. The theorem to be proved is then negated and an attempt is made to derive a contradiction. The next stage is to eliminate any variable globally defined by an equality, eliminate special functions such as abs , mod , and div , break the problem into subproblems, and call the linear prover. The linear prover deals with linear systems of inequalities and tries to find a contradiction or at least narrow down the range of values of the variables. These results are then applied to the remaining (if

any) non-linear relations in an attempt to derive a contradiction. King reports that the time taken for the simplification of most theorems took about 10 seconds per theorem, the only complicated part of his system being the linear prover.

Another decision procedure designed specifically for proving theorems in a program verifier is described by Nelson and Oppen [Nelson and Oppen 79]. They describe how to combine decision procedures for four different quantifier-free theories: the theory of real numbers under $+$ and \leq , the theory of arrays under **store** and **select**, the theory of list structure with **car**, **cdr**, **cons** and **atom**, and the theory of equality with uninterpreted function symbols. Each theory is characterized by its set of nonlogical symbols and nonlogical axioms. These four theories are combined by propagating any equalities entailed in each theory to the decision procedures of the other theories. An algorithm is described for the equality propagation procedure and a proof of its correctness is given. The resulting decision procedure is NP-complete. A shortcoming of this decision procedure is the fact that only multiplication by constants can be handled (e.g. $2 * x$ can be written as $x + x$). This system is a part of the Stanford Pascal Verifier, an interactive system for reasoning about Pascal programs.

A suggestion regarding a method of building a special-purpose theorem prover for verifying programs has been put forward in [Sarkar and De Sarkar 89a]. According to the authors, it is counter-intuitive to use resolution-based theorem provers for proving verification conditions arising during the verification of programs over integers, since this requires translating the conditions to be proved into predicate calculus, while for program verification it is more natural for the user to provide assertions in algebraic notation. They present a new inference rule, called implication-resolution, which is a generalization of resolution. This and other inference rules can be applied directly to the formulas of integer arithmetic. Every term is expressed in a normal form (this is the same normal form as described in [King 69]), and then the various inference rules are applied. In a related paper [Sarkar and De Sarkar 89b], Sarkar and De Sarkar describe a set of inference rules for handling quantified formulas and arrays in verifying integer programs. The integer axioms are built into the inference rules rather than being provided as premises. A normal form is described for quantified formulas. This paper does not treat existentially quantified formulas. The rules described in the above two papers have been implemented in a theorem prover for proving the verification conditions arising in iterative programs over integers [Sarkar and De Sarkar 89c]. The prover is written in Pascal and implemented on a HP 9000 minicomputer. An assessment of the efficiency of the prover is given based on the efficiency of proof construction and the memory space used.

Spitzen and Wegbreit [Spitzen and Wegbreit 75] have done some work on the

unification and synthesis of data structures. They discuss how data structures can be precisely specified and give examples using data structures like stacks, buffers, and queues. They present an axiomatization of a programming language suitable for automatic verification, and show how programs which realize these data structures may be proved correct. Spitzen and Wegbreit believe that whereas mechanical verification is within the reach of current verification systems, mechanical synthesis is substantially harder since it seems to be inherently a second-order process requiring some form of induction.

3.2 Floyd's inductive assertions method

In [Floyd 67], Floyd describes a method for verifying flowchart programs. (This description is taken from [Manna 74].) Suppose we are given a flowchart program with a description of its behavior, i.e. a characteristic predicate ψ (called an output predicate), which describes the relationships among the program variables that must be satisfied at the completion of the program execution. We are also given an input predicate ϕ , which defines the input restrictions that must be satisfied to make execution of the program meaningful. Our task is to guarantee that for all program executions with inputs satisfying the input predicate, the program terminates, and that at the completion of execution the output predicate is satisfied.

We distinguish among three types of variables, written as three vectors :

- (i) an input vector $\bar{x} = (x_1, x_2, \dots, x_n)$, which consists of the given input values and therefore never changes during computation;
- (ii) a program vector $\bar{y} = (y_1, y_2, \dots, y_m)$, which is used as temporary storage during computation; and
- (iii) an output vector $\bar{z} = (z_1, z_2, \dots, z_l)$, which yields the output values when computation terminates.

We say that a program is partially correct with respect to ϕ and ψ if for every input ξ such that $\phi(\xi)$ is true and the computation of the program terminates, ψ is true for the values of the program variables at the completion of execution. Thus in partial correctness we don't care about termination.

Suppose we are given a flowchart program P , an input predicate ϕ , and an output predicate ψ . To prove that P is partially correct with respect to ϕ and ψ we proceed as follows :

1. **Cutpoints.** The first step is to cut the loops of the program by choosing on the arcs of the flowchart a finite set of points, called cutpoints, such that every loop includes at least one such cutpoint.

2. Inductive assertions. The next step is to associate with each cutpoint i of the program a predicate $p_i(\bar{x}, \bar{y})$, called the inductive assertion (or loop invariant), which characterizes the relation among the variables at that cutpoint. In other words, whenever control reaches point i , $p_i(\bar{x}, \bar{y})$ must be true for the current values of \bar{x} and \bar{y} at this point. The input predicate $\phi(\bar{x})$ is attached to the START point, and the output predicate $\psi(\bar{x}, \bar{z})$ is attached to the HALT points.

3. Verification conditions. The third step is to construct for every path α leading from cutpoint i to cutpoint j the verification condition

$$\forall \bar{x} \forall \bar{y} [p_i(\bar{x}, \bar{y}) \wedge R_\alpha(\bar{x}, \bar{y}) \rightarrow p_j(\bar{x}, r_\alpha(\bar{x}, \bar{y}))]$$

where $R_\alpha(\bar{x}, \bar{y})$ indicates the condition for path α to be traversed, and $r_\alpha(\bar{x}, \bar{y})$ describes the transformation of the values of \bar{y} effected while path α is traversed. A backward-substitution technique for obtaining R_α and r_α is as follows. Let α be a path leading from cutpoint i to j . Initially, $R(\bar{x}, \bar{y})$ is set to *true* and $r(\bar{x}, \bar{y})$ is set to \bar{y} , and both are attached to cutpoint j ; then in each step, the old R and r are used to construct the new R and r , moving backward toward cutpoint i . The final R and r obtained at cutpoint i are the desired R_α and r_α . The rules for constructing the new R and r in each step are given below, according to the statement occurring just before the old R and r .

1. Statement : $\bar{y} \leftarrow f(\bar{x})$
New values for R and r : $R(\bar{x}, f(\bar{x})), r(\bar{x}, f(\bar{x}))$.
2. Statement : $\bar{y} \leftarrow g(\bar{x}, \bar{y})$
New values for R and r : $R(\bar{x}, g(\bar{x}, \bar{y})), r(\bar{x}, g(\bar{x}, \bar{y}))$.
3. Statement : $t(\bar{x}, \bar{y})$ (test condition, where the old R and r are on the "true" branch leading out of the condition)
New values for R and r : $t(\bar{x}, \bar{y}) \wedge R(\bar{x}, \bar{y}), r(\bar{x}, \bar{y})$.
4. Statement : $t(\bar{x}, \bar{y})$ (test condition, where the old R and r are on the "false" branch leading out of the condition)
New values for R and r : $\neg t(\bar{x}, \bar{y}) \wedge R(\bar{x}, \bar{y}), r(\bar{x}, \bar{y})$.

4. Proving the verification conditions. The fourth and final step is to prove that all these verification conditions for our choice of inductive assertions are true. Proving the verification conditions implies that each predicate attached to a cutpoint has the property that whenever control reaches the point, the predicate is true for the current values of the variables; in particular, whenever control reaches a HALT point, $\psi(\bar{x}, \bar{z})$ is true for the current values of \bar{x} and \bar{z} . In other words, proving the verification conditions shows that the given program is partially correct with respect to ϕ and ψ .

All of these steps are rather mechanical except for Step 2. Discovering the

proper loop invariants to attach to cutpoints is a non-trivial matter and requires a thorough understanding of the program.

3.3 Overview of the method

In this section we give an overview of an iteration method to derive loop invariants for programs for the purpose of program verification. Suppose we are given a program. Perform the following steps :

1. Draw a flowchart for the program, cut the loops, and attach loop invariants (these are unknown) and input and output assertions where appropriate. Note that we are assuming that loop invariants exist for all loops; if they do not, then this method is not applicable. A symbol is attached at every loop cutpoint; this symbol represents an unknown loop invariant.

2. Generate the verification conditions for the program as explained in the previous section.

3. Apply the iteration method to the formulas of the verification conditions to obtain the loop invariants.

Step 3 needs to be described in detail. We give below a brief overview of the method we will use. The detailed algorithm is given in Section 3.6.

Note that a “known” formula is one which does not contain any loop invariant. In the following, W , W_1 and W_2 denote loop invariants, and H , H_1 and H_2 denote known formulas. Any verification condition involving a loop invariant is of one of the following three forms.

$$(i) H \rightarrow W$$

$$(ii) H \wedge W_1 \rightarrow W_2$$

$$(iii) H_1 \wedge W \rightarrow H_2.$$

To see that this is true, recall that there is one cutpoint for every loop in the program, one cutpoint at the entry of the program, and one cutpoint at every exit of the program. Therefore a path in the program could be of one of the following four types :

- (1) A path from the entry cutpoint to a loop cutpoint
- (2) A path from a loop cutpoint to a loop cutpoint
- (3) A path from a loop cutpoint to an exit cutpoint
- (4) A path from the entry cutpoint to an exit cutpoint

Of these four types, a verification condition for a path of type 4 does not involve any loop invariants and will therefore not be considered here. A verification condition for a path of type 1 will be of the form $H \rightarrow W$ (where H is a known

formula and W is the loop invariant at the cutpoint at the end of the path), since the conditions which hold at the beginning of the path and during the path traversal are known, and W is the loop invariant of the cutpoint at the end of the path. A verification condition for a path of type 2 will be of the form $H \wedge W_1 \rightarrow W_2$ (where H is a known formula and W_1, W_2 are the loop invariants of the cutpoints at the beginning and end of the path respectively), since the condition which holds at the beginning of the path is W_1 , the conditions which hold during the path traversal are known, and W_2 is the loop invariant of the cutpoint at the end of the path. A verification condition for a path of type 3 will be of the form $H_1 \wedge W \rightarrow H_2$ (where H_1, H_2 are known formulas and W is the loop invariant of the cutpoint at the beginning of the path), since the condition which holds at the beginning of the path is W , the conditions which hold during the path traversal are known, and the output condition which holds at the end of the path is known.

We will obtain successively more accurate approximations to the loop invariants. For this purpose, we will define the function GET-APPROX in Section 3.7 to be a binary function which takes as arguments a formula H and a symbol W and returns a formula F such that $H \rightarrow F$ which is an approximation for W . Note that H must be a known formula; W is the name of an unknown loop invariant.

Initially, only the input and output assertions are given. We initially approximate all the loop invariants by setting them to "false". We will represent the i^{th} approximation to W by W_i ; the initial approximation to W is W_0 . Informally, the method we will use is the following : suppose W is some (unknown) loop invariant in the program. Consider all the verification conditions in which W appears on the right-hand side of the implication sign. We replace all occurrences of loop invariants in these verification conditions with their current approximations. Suppose that the last approximation calculated for W was W_i . Suppose the resulting verification conditions are :

$$H_1 \rightarrow W_i$$

$$H_2 \rightarrow W_i$$

$$H_3 \rightarrow W_i$$

· ·

· ·

· ·

$$H_n \rightarrow W_i$$

(where i gives the number of the current iteration). Note that loop invariants may occur in the formulas H_1, H_2, \dots, H_n above; all such occurrences are replaced by the current approximations for these loop invariants. For all the H_j 's, $1 \leq j \leq n$, check whether $H_j \rightarrow W_i$ is true or not. Let T be the set of all H_j 's such that $H_j \rightarrow W_i$

is not true. If T is empty, then set $W_{i+1} = W_i$; if T is not empty, then set $W_{i+1} = \text{GET-APPROX}(W_i \vee R, W)$, where $R = \bigvee \{H_j \mid H_j \in T\}$. Note that for all j such that $1 \leq j \leq n$, $H_j \rightarrow W_{i+1}$ and $W_i \rightarrow W_{i+1}$. This is because the new formula W_{i+1} generated by the function GET-APPROX is a logical consequence of the disjunction of W_i and all the formulas in the set T ; thus it is a logical consequence of each of these formulas.

We then look for another loop invariant and find the next approximation to it exactly as described for W_i (this time using W_{i+1} as an approximation for W), and so on. Recall that all the verification conditions in which W appears on the right-hand side of the implication sign are $H_1 \rightarrow W, H_2 \rightarrow W, H_3 \rightarrow W, \dots, H_n \rightarrow W$. If we have $W_{i+1} = W_i$ (this happens when the set T is empty), then since $H_j \rightarrow W_{i+1}$ for all j such that $1 \leq j \leq n$, and since $W_{i+1} = W_i$, we have $H_j \rightarrow W_i$ for all j such that $1 \leq j \leq n$. When this happens for all the inductive assertions, then we are done.

3.4 Some observations about the programming language model

A program is partially correct if all the verification conditions derived from the program after assigning appropriate loop invariants are valid in a model \mathcal{M} of the data structures and primitive operations of the language. For instance, most programming languages contain the arithmetic operators $+$ and $-$; hence a model \mathcal{M} of such a programming language would reflect the semantics of these operations. In other words, we would like to prove

$$\mathcal{M} \models vc$$

for all the verification conditions “ vc ” of the program being verified.

Now the question of whether there exists an axiomatization of the model \mathcal{M} arises. (A theory T is said to be *axiomatizable* if there exists a decidable set $W \subseteq T$ such that T is exactly the set of all formulas derivable from W in the predicate calculus.) Some examples of axiomatizable theories are the set of all logically valid first-order formulas, the theory of natural numbers with successor function, and Presburger arithmetic (Presburger arithmetic consists of addition and the predicate “ $<$ ”, over the natural numbers). Peano arithmetic (addition and multiplication along with the predicate “ $<$ ” over the natural numbers) is not axiomatizable; however, the well-known Peano axioms along with the principle of induction over the natural numbers characterize all properties of the natural numbers, including those of the Peano arithmetic, i.e. those which may be expressed as formulas of

first-order logic. This is not in contradiction to the fact that Peano arithmetic is not axiomatizable, because the principle of induction cannot be expressed in first-order logic, as it involves quantification over predicates [Loeckx and Sieber 87].

Assuming that there exists an axiomatization A of a model \mathcal{M} (A is a set of axioms such that A is decidable and such that the set of all formulas true under \mathcal{M} is exactly the set of all formulas which are derivable from A in the predicate calculus), the verification problem reduces to a proof of the form

$$A \vdash vc$$

for all verification conditions “ vc ” of the program being verified. Now, any verification condition is of the form $L \rightarrow M$ (see Section 3.2), for first-order formulas L and M . Therefore the above can be written as

$$A \vdash (L \rightarrow M)$$

which is equivalent to

$$\vdash ((A \wedge L) \rightarrow M)$$

Henceforth we will assume that the models of the programming languages under consideration are axiomatizable, and that an axiomatization A of the language is provided when the verification conditions are being proved; in other words, the formulas in A are taken to be axioms and can be used for any proof. This may seem like a very restrictive assumption, since we know that even Peano arithmetic is not axiomatizable; however, in many cases, we circumvent this problem by providing suitable instances of the principle of induction as required, or by providing the system with enough facts to be able to derive the desired formulas from these facts. The reader should nevertheless be aware of this restriction on the power of our system.

3.5 Description of algorithm for generating loop invariants

The notation described in Section 3.3 is used throughout this algorithm. We first briefly describe the algorithm step by step. We assume that W^1, W^2, \dots, W^n are the loop invariants of the program. As mentioned previously, we let the initial approximations of all verification conditions be “false”, and denote the initial approximation of each W^i by W_0^i . The j^{th} approximation for W^i will be denoted by W_j^i for every i , $1 \leq i \leq n$. If the last approximation which has been calculated for a loop invariant W^i is the k^{th} approximation, then “ $index(W^i)$ ” is set to “ k ”. Initially, $index(W^i) = 0$ for every i , $1 \leq i \leq n$. These initializations are performed in step 1.

In step 2, we construct a list called “*list*”, containing all the loop invariants’ names (i.e. W^1, W^2, \dots, W^n) in a particular order. The order is decided as follows. First, all loop invariants which figure on right-hand sides of implications of verification conditions of the form $H \rightarrow W^i$ are appended to the list (in any order). Note that no duplicate elements are ever permitted in the list; if some loop invariant is already in the list, it is not added to the list again. We then start examining the list, starting at the head of the list, and keep adding new elements at the tail of the list depending on what the element of the list being examined is. This is done as follows. Suppose the element at the head of the list is W^k . Then we look for all verification conditions which contain W^k on the left-hand side of their implication sign, and which contain some loop invariant W^i on the right-hand side of their implication sign, and for every such verification condition, if W^i is not already a member of the list, it is added at the tail of the list. This process is then repeated for the second element of the list, the third, and so on, until every loop invariant has been added to the list.

The list built in step 2 is used in step 3 to provide the order in which the iteration will proceed. Starting with the first loop invariant in this list, and repeating the same process for each element of the list in order, we do the following. Initialize the set T to be an empty set. Suppose that the first element in *list* is the loop invariant W . We go through the list of all the verification conditions, and for every verification condition which has W on the right-hand side of its implication sign, we do the following. Suppose this verification condition is $J \rightarrow W$. J could either be of the form $J = H$, for some known formula H , or J could be of the form $J = H \wedge W^j$, for some loop invariant W^j . Note that W^j could be equal to W . If J is of the latter form, then W^j is replaced by $W_{index(W^j)}^j$ in J . Call the transformed formula “ J' ”. We then check whether $J' \rightarrow W_{index(W)}$ is true or not. If it isn’t, we add J' to the set T . This process is repeated for all the verification conditions. We then obtain the next approximation for W as follows. First, $index(W)$ is incremented by 1. If the set T is empty, then the current approximation for W is retained, and “ $flag(W)$ ” is set to “true” to mark this fact. If the set T is not empty, then we obtain the next approximation $W_{index(W)}$ for W by calling the routine “GET-APPROX” to return a formula $W_{index(W)}$ such that

$$W_{index(W)-1} \vee R \rightarrow W_{index(W)},$$

where R is the disjunction of all the elements of T . The fact that T was non-empty is marked by setting $flag(W)$ to “false”.

This whole process is repeated until all the flags for all the loop invariants are set to “true” at the same time. This indicates that the current approximations for the loop invariants satisfy all the verification conditions and can therefore be used

as valid loop invariants. Step 4 sets W_{approx}^i to be the last approximation obtained for W^i , which is a valid loop invariant for every i such that $1 \leq i \leq n$.

The algorithm is given below in Pascal-like pseudo-code.

3.6 The iteration algorithm

{ COMMENT : Let V be the set of verification conditions for the program; suppose $V = \{vc_1, vc_2, \dots, vc_m\}$. Let all the (unknown) loop invariants be W^1, W^2, \dots, W^n . We denote the formula on the left-hand side of the implication sign in a verification condition “ vc ” by “ $lhs(vc)$ ”, and similarly we denote the formula on the right-hand side of the implication sign in a verification condition “ vc ” by “ $rhs(vc)$ ”.

```

1. For  $i := 1$  to  $n$  do
    {  $index(W^i) := 0$ ;
       $W_0^i := false$ 
    }
2.  $list := empty$ ;
    $ptr := 1$ ;
   for  $i := 1$  to  $m$  do
       if ( $vc_i$  is of the form  $H \rightarrow W^i$ ) and not(member( $list, W^i$ )) then
           append( $list, W^i$ );
   while length( $list$ ) <  $n$  do
       {  $current := ptr^{th}$  element in  $list$ ;
         for every  $vc \in V$  such that ( $lhs(vc)$  contains  $current$ ) and ( $rhs(vc) = W^i$ 
         for some  $i$ ) and (not(member( $list, W^i$ ))) do
             append( $list, W^i$ );
          $ptr := ptr + 1$ 
       }
3. repeat
    for  $i := 1$  to  $n$  do
        {  $W := i^{th}$  element in  $list$ ;
           $T := \emptyset$ ;
          for  $j := 1$  to  $m$  do
              { if  $rhs(vc_j) = W$  then
                  if  $\neg$  ( $lhs(vc_j)$  implies  $W_{index(W)}$ ) then
                       $T := T \cup \{ lhs(vc_j) \}$ 
              };

```

```

    index(W) := index(W) + 1;
    if T ≠ ∅ then
        { flag(W) := false;
          Windex(W) := GET-APPROX(Windex(W)-1 ∨
            (∨j{H|H ∈ T}), W) (see Note after algorithm)
        }
    else
        { flag(W) := true;
          Windex(W) := Windex(W)-1
        }
    }
until ∧i=1n flag(Wi);

```

4. for $i := 1$ to n do

$$W_{approx}^i := W_{index(W)}^i .$$

5. Halt.

Note. Each time GET-APPROX is called, any occurrence of an unknown loop invariant W^j in the first argument of GET-APPROX is replaced by its current approximation, which is $W_{index(W^j)}^j$.

We now describe the function GET-APPROX. We will then prove that this iteration algorithm is sound and complete.

3.7 The function GET-APPROX

The function GET-APPROX takes two arguments H and W , where H is a known formula and W is the name of a loop invariant for which GET-APPROX will return an approximation. Note that the value of W is unknown. We will see that GET-APPROX can return a formula F such that $H \rightarrow F \rightarrow W$ and such that $F \preceq W$. In Chapter 2, we saw that such a formula can be derived by resolution from $Sk(H)$ and unskolemization.

The derivation of F can be made more efficient by noting that the problem at hand is really simpler than just deriving logical consequences of one formula. To see this, note that the argument H is a disjunction

$$H = W_i \vee H_1 \vee H_2 \vee \dots \vee H_k$$

where W_i is the previous approximation obtained for W , and each H_j ($1 \leq j \leq k$) is the left-hand side of a verification condition for which the right-hand side is W , and which is not valid with the current approximations for loop invariants (see the iteration algorithm). Note that k could be zero here.

Our goal is to generate a formula S which is a logical consequence of H , i.e. such that

$$W_i \vee H_1 \vee H_2 \vee \dots \vee H_k \rightarrow S.$$

The above implication is equivalent to the $k + 1$ implications

$$W_i \rightarrow S$$

$$H_1 \rightarrow S$$

.

.

.

$$H_k \rightarrow S.$$

We therefore need to generate a formula S which is a logical consequence of each of W_i, H_1, \dots, H_k . We have $k + 1$ formulas, each of which imply S , and we are trying to derive S from these. We can generate a logical consequence for each of W_i, H_1, \dots, H_k by the resolution and unskolemization method of Chapter 2. Suppose a logical consequence F of one of these formulas has been generated by resolution and then unskolemization. We then check if F is implied by all of the formulas W_i, H_1, \dots, H_k (it is obviously implied by at least one of them, since F is a logical consequence of one of these formulas). Such a formula F is obtained for each of the formulas W_i, H_1, \dots, H_k . We then collect together the F 's which are implied by all of the formulas W_i, H_1, \dots, H_k (i.e. $W_i \rightarrow F, H_1 \rightarrow F, H_2 \rightarrow F, \dots, H_k \rightarrow F$) and let S be their conjunction. S is then returned by GET-APPROX as the $i + 1^{\text{th}}$ approximation for W . Clearly, $H \rightarrow S$, since each F in the conjunction S was implied by all of the formulas W_i, H_1, \dots, H_k .

If after a number " b " of trials, we are not able to obtain any F which is derived from one of W_i, H_1, \dots, H_k and is implied by all of them, then we take S to be the disjunction of all the $k + 1$ formulas each of which was a logical consequence of one of W_i, H_1, \dots, H_k . Here too, $H \rightarrow S$. In the algorithm, " b " is a bound input by the user.

The approach is slightly different when an approximation is being generated for a loop invariant W for which there exists at least one verification condition of the form $H' \wedge W \rightarrow H''$, where H' and H'' are known formulas (i.e. W appears on the left-hand side of a verification condition whose right-hand side is a known formula). In this case, we adopt an approach which guides the search for a loop invariant more effectively than that described above. Here, before generating logical consequences of W_i, H_1, \dots, H_k , we first check whether $H' \wedge H \rightarrow H''$ is valid for all verification conditions of the form $H' \wedge W \rightarrow H''$ (if this is not the case, we backtrack). If $H' \wedge H \rightarrow H''$ is valid for all verification conditions of the form $H' \wedge W \rightarrow H''$, this means that

$$H' \wedge (W_i \vee H_1 \vee \dots \vee H_k) \rightarrow H''$$

is valid for all such verification conditions. Therefore

$$H' \wedge W_i \rightarrow H''$$

$$H' \wedge H_1 \rightarrow H''$$

.

.

.

$$H' \wedge H_k \rightarrow H''$$

are all valid.

Recall that in the preceding paragraphs, we generated logical consequences of each of $W_i, H_1, H_2, \dots, H_k$ in our search for an approximation for W . In this case, however, we have one more piece of information about W , namely :

$$H' \wedge W \rightarrow H''.$$

Therefore, an approximation F for W must satisfy the above formula when substituted for W , i.e. we must have

$$H' \wedge F \rightarrow H''.$$

We therefore proceed as follows. Let B_j be any one of W_i, H_1, \dots, H_k , and suppose that $H' \wedge W \rightarrow H''$ is a verification condition. Since $H' \wedge B_j \rightarrow H''$ is valid, $H' \wedge B_j \wedge \neg H''$ is unsatisfiable in the model of the programming language being used. Therefore there exists a resolution proof of the unsatisfiability of $H' \wedge B_j \wedge \neg H''$ in this model (by the completeness of resolution). Recall that from the discussion in Section 3.4, a set *AXIOMS* of axioms which characterize the programming language model are to be used in this resolution proof. Consider some derivation of the empty clause from $Sk(AXIOMS \wedge H' \wedge B_j \wedge \neg H'')$. We can perform as many of the resolutions in this derivation as possible between $Sk(B_j \wedge AXIOMS)$ first, and then perform resolutions with the resulting clauses and $Sk(H' \wedge \neg H'')$. Consider the set of clauses (called PROOFS, say) thus derived from $Sk(B_j \wedge AXIOMS)$. From the above, $PROOFS \wedge Sk(H' \wedge \neg H'')$ is unsatisfiable. We therefore unskolemize some subset of PROOFS to obtain a formula F . After such an F is obtained for each of W_i, H_1, \dots, H_k , we proceed as explained earlier and obtain a formula S as before. Note that S is a logical consequence of each of $W_i, H_1, H_2, \dots, H_k$, and hence of H . We then check whether $H' \wedge S \rightarrow H''$ is valid for all verification conditions of the form $H' \wedge W \rightarrow H''$ (if this is not the case, we backtrack). As noted earlier, this is a necessary condition for S to be a valid approximation for W .

The method described above restricts the search for an approximation S for W to all possible sets "PROOFS" generated as explained above, rather than the set of all possible logical consequences of H . The following discussion shows that it is sufficient to do so. We know that a valid approximation F for W can be generated

as a logical consequence of B_j , i.e. $Sk(F)$ can be generated by resolution from $Sk(B_j \wedge AXIOMS)$. Suppose such an approximation F has been generated. Since F is a valid approximation for W , it must satisfy

$$H' \wedge F \rightarrow H''.$$

Hence we can obtain a resolution proof of the unsatisfiability of $Sk(AXIOMS) \wedge Sk(H') \wedge Sk(F) \wedge Sk(\neg H'')$. However, $Sk(F)$ was obtained by resolution from $Sk(B_j \wedge AXIOMS)$. Therefore it is possible to obtain $Sk(F)$ by resolution from $Sk(B_j \wedge AXIOMS)$ in a resolution proof of the unsatisfiability of

$$Sk(AXIOMS) \wedge Sk(B_j) \wedge Sk(H') \wedge Sk(\neg H''),$$

since such a resolution proof can be obtained by :

1. Deriving $Sk(F)$ from $Sk(B_j \wedge AXIOMS)$
2. Deriving the empty clause from $AXIOMS \wedge Sk(F) \wedge Sk(H') \wedge Sk(\neg H'')$.

This method of generating F helps to restrict the search for a loop invariant, since the known formulas H'' on the right-hand sides of verification conditions of the form $H' \wedge W \rightarrow H''$ help to direct the search for F . This will be more clearly demonstrated in the examples in Section 3.11.

Note. W_i may itself be a disjunction of formulas, i.e. we may have

$$W_i = A_1 \vee A_2 \vee \dots \vee A_m.$$

In this case, $H = A_1 \vee \dots \vee A_m \vee H_1 \vee \dots \vee H_k$; thus we will have $k + m$ formulas for which logical consequences have to be generated (unlike the above-described situation where we had $k + 1$ such formulas).

The algorithm for the function GET-APPROX and two procedures which it calls are given below.

function GET-APPROX(H, W);

begin

 input(b);

 if there is one or more verification condition of the form $H_1 \wedge W \rightarrow H_2$ then

$S :=$ DIRECTED_SEARCH(H, W)

 else $S :=$ CONSEQUENCE(H, W);

 return(S)

end.

function CONSEQUENCE(H, W);

begin

$S :=$ true;

 num.iterations := 0;

```

Suppose  $H = B_1 \vee B_2 \vee \dots \vee B_r$ ;
while ( $S = true$ ) and ( $num\_iterations < b$ ) do
  { $num\_iterations := num\_iterations + 1$ ;
  for  $i := 1$  to  $r$  do
    generate a formula  $S_i \in unsk(Res(B_i \wedge AXIOMS))$ ;
  for  $i := 1$  to  $r$  do
    if  $B_k \rightarrow S_i$  for all  $k$  such that  $1 \leq k \leq r$  then
       $S := S \wedge S_i$ 
    }
  }
if ( $S = true$ ) then let  $S = S_1 \vee S_2 \vee \dots \vee S_r$ ;
return( $S$ )

```

end.

function DIRECTED_SEARCH(H, W);

begin

```

check if  $H_1 \wedge H \rightarrow H_2$  for all verification conditions of the form  $H_1 \wedge W \rightarrow H_2$ ; if not, then BACKTRACK;
num_iterations := 0;
 $S := true$ ;
Suppose  $H = B_1 \vee B_2 \vee \dots \vee B_r$ ;
while ( $S = true$ ) and ( $num\_iterations < b$ ) do
  { $num\_iterations := num\_iterations + 1$ ;
  for  $i := 1$  to  $r$  do
    {PROOFS := set of all clauses generated from  $B_i \wedge AXIOMS$  by resolution during some proof of  $H_1 \wedge B_i \rightarrow H_2$ ; (see Note 1 below)
    choose  $S_i \in unsk(SUB)$  for some subset  $SUB$  of PROOFS
    }
  for  $i := 1$  to  $r$  do
    if  $B_k \rightarrow S_i$  for all  $k$  such that  $1 \leq k \leq r$  then
       $S := S \wedge S_i$ 
    }
  }
if ( $S = true$ ) then let  $S = S_1 \vee S_2 \vee \dots \vee S_r$ ;
check if  $H_1 \wedge S \rightarrow H_2$  for all verification conditions of the form  $H_1 \wedge W \rightarrow H_2$ ;
if not, then BACKTRACK;
return( $S$ )

```

end.

Note 1. Since $H_1 \wedge H \rightarrow H_2$ is valid for all verification conditions of the form $H_1 \wedge W \rightarrow H_2$, and since $H = B_1 \vee \dots \vee B_r$, clearly for every i such that $1 \leq i \leq r$, $H_1 \wedge B_i \rightarrow H_2$. Consider a proof of unsatisfiability of $Sk(AXIOMS \wedge H_1 \wedge B_i) \wedge Sk(\neg H_2)$ by resolution, for one such verification condition $H_1 \wedge W \rightarrow H_2$. These resolutions can be rearranged so that any resolutions among clauses of B_i and clauses of $AXIOMS$ are performed first. Let the set of clauses from B_i and $AXIOMS$ and the clauses generated by these resolutions be the set PROOFS.

3.8 Proof of completeness of the iteration algorithm

We now prove the completeness of this algorithm, i.e. we show that if there exists a valid loop invariant for a given loop, the algorithm can derive it. The proof of completeness is based on the following five facts :

- (i) The first time that GET-APPROX(H, W) is called, $H \rightarrow W$ is valid.
- (ii) If $H \rightarrow W$, then GET-APPROX(H, W) can return S such that $S \preceq W$.
- (iii) If GET-APPROX(H, W) has returned S such that $S \preceq W$ all the n times it has been called, then when it is called for the $n + 1^{th}$ time, H will imply W .
- (iv) GET-APPROX(H, W) can always return S such that $S \preceq W$.
- (v) If GET-APPROX always returns a formula such that $S \preceq W$, where W is its second argument, then the algorithm terminates and returns a valid loop invariant.

We prove these statements one by one.

Proof of (i) : Suppose GET-APPROX has not yet been called. Then the approximations for all loop invariants are currently set to “*false*”. Consider the first argument H of GET-APPROX. If the second argument of GET-APPROX is W , then H is a disjunction of W_0 and the left-hand sides of verification conditions which have W on their right-hand sides and which are not valid with W set to “*false*”. However, it can be seen that none of these left-hand sides can contain an occurrence of any loop invariant; the reason for this is that since all the current approximations for all loop invariants are “*false*”, any left-hand side containing an occurrence of a loop invariant would have the value “*false*” (since $false \wedge H' \equiv false$ for all H'). And since $false \rightarrow X$ is valid no matter what the value of X is, a verification condition containing a loop invariant in its left-hand side would be valid. Thus all the left-hand sides of verification conditions which are included as disjunctions in H must be known formulas without any occurrences of loop invariants, and the right-hand sides of these verification conditions are all W . Hence each of these left-hand sides must imply W . Since $W_0 \rightarrow W$ (because $W_0 = false$), clearly here $H \rightarrow W$.

Proof of (ii) : Suppose GET-APPROX(H, W) is called, and suppose $H \rightarrow W$. GET-APPROX in turn calls either CONSEQUENCE or DIRECTED_SEARCH.

Case (1). Suppose CONSEQUENCE(H, W) gets called by GET-APPROX. We have $H = B_1 \vee B_2 \vee \dots \vee B_r$, where some of the B_i 's constitute the previous approximation for W , and the remaining B_j 's are left-hand sides of verification conditions whose right-hand sides are W . Now, since $H \rightarrow W$, we have

$$B_1 \vee B_2 \vee \dots \vee B_r \rightarrow W$$

Therefore for any i ,

$$B_i \rightarrow (B_1 \vee B_2 \vee \dots \vee B_r) \rightarrow W$$

i.e. $(B_i \rightarrow W)$ is valid for all $1 \leq i \leq r$.

Since for each i , an S_i can be found in $unsk(Res(B_i \wedge AXIOMS))$ with the property that $S_i \preceq W$ (because $B_i \rightarrow W$; thus we can use the results from Chapter 2), it is possible to obtain the S_i 's above so that $S_i \preceq W$ for all $1 \leq i \leq r$. It may happen that S_i is implied by some of the other B_k 's; if it is implied by all the B_k 's, then S_i is added as a conjunct in formula S . After this is done for each i such that $1 \leq i \leq r$, S will be a conjunction of S_i 's such that $S_i \preceq W$ for all i . But then by Theorem 2.6, $S \preceq W$.

If not even one formula S_i can be derived from B_i such that S_i is implied by all the other B_j 's (for any i such that $1 \leq i \leq r$) in a number b of trials, then S is taken to be the disjunction of the last set of S_i 's which were obtained in the WHILE loop; since each of these S_i 's had the property that $S_i \preceq W$, by Theorem 2.6, we have $S \preceq W$.

Hence we see that it is possible for the function CONSEQUENCE to return a formula S which has the property that $S \preceq W$. Since GET-APPROX also returns S , this case is proved.

Case (2). Suppose DIRECTED_SEARCH(H, W) gets called by GET-APPROX. We have $H = B_1 \vee B_2 \vee \dots \vee B_r$, where some of the B_i 's constitute the previous approximation for W , and the remaining B_j 's are left-hand sides of verification conditions whose right-hand sides are W . As in Case (1), since $H \rightarrow W$, we get

$$B_i \rightarrow W \text{ for all } i \text{ such that } 1 \leq i \leq r.$$

Now, since DIRECTED_SEARCH has been called, there exist verification conditions of the form $H_1 \wedge W \rightarrow H_2$, where W is the second argument of DIRECTED_SEARCH and where H_1, H_2 are known formulas. Note that since $H \rightarrow W$, we have $H_1 \wedge H \rightarrow H_1 \wedge W \rightarrow H_2$, i.e. $H_1 \wedge H \rightarrow H_2$ is valid and therefore the condition to be checked at the entry to the function holds. Let $H_1 \wedge W \rightarrow H_2$ be one such verification condition.

Since $B_i \rightarrow W$ for all i such that $1 \leq i \leq r$, we have $H_1 \wedge B_i \rightarrow H_1 \wedge W \rightarrow H_2$, i.e. $H_1 \wedge B_i \rightarrow H_2$. For any i , consider the set of clauses $Sk(B_i \wedge AXIOMS \wedge H_1) \wedge$

$Sk(\neg H_2)$ ($1 \leq i \leq r$). This set of clauses is unsatisfiable and therefore there exist derivations of the empty clause from these clauses. Now, it is possible to derive a set of clauses \mathcal{D}_i from $Sk(B_i \wedge AXIOMS)$ and to unskolemize \mathcal{D}_i to give a formula S_i such that

$$B_i \rightarrow S_i \rightarrow W \text{ and } S_i \preceq W \text{ (from the theorems in Chapter 2).}$$

Then, since $S_i \rightarrow W$, therefore $H_1 \wedge S_i \rightarrow H_2$ is valid; hence since by Theorem 2.2, $(\forall \mathcal{D}_i) \rightarrow S_i$ is valid, therefore $H_1 \wedge (\forall \mathcal{D}_i) \rightarrow H_1 \wedge S_i \rightarrow H_2$, i.e. $H_1 \wedge (\forall \mathcal{D}_i) \rightarrow H_2$ is valid; therefore there exists a derivation of the empty clause from $Sk(H_1 \wedge AXIOMS) \wedge \mathcal{D}_i \wedge Sk(\neg H_2)$.

Thus we see that there exists a derivation of the empty clause from $Sk(B_i \wedge AXIOMS) \wedge Sk(H_1) \wedge Sk(\neg H_2)$ such that a set \mathcal{D}_i of clauses is produced by resolution from $Sk(B_i \wedge AXIOMS)$ during this derivation of the empty clause such that \mathcal{D}_i has the above-mentioned properties (namely that \mathcal{D}_i can be unskolemized to give S_i such that $S_i \preceq W$).

Therefore if we let PROOFS be defined as in Note 1 at the end of the function DIRECTED_SEARCH, for such a derivation of the empty clause, it is possible to pick a set of clauses \mathcal{D}_i from PROOFS such that for some $S_i \in unsk(\mathcal{D}_i)$, $S_i \preceq W$. As noted in Section 3.7, this method directs the search for W . This will be demonstrated in examples to follow. Thus for each i , it is possible to choose the S_i 's above so that $S_i \preceq W$ for all $1 \leq i \leq k$. It may happen that S_i is implied by some of the other B_k 's; if it is implied by all the B_k 's, then S_i is added as a conjunct in formula S . After this is done for all i such that $1 \leq i \leq r$, S will be a conjunction of S_i 's such that $S_i \preceq W$ for all i . But then by Theorem 2.6, $S \preceq W$.

If no such formulas S_i , such that S_i is implied by all the B_k 's, can be derived from the B_i 's in a number b of trials, then S is taken to be the disjunction of the last set of S_i 's which were obtained in the WHILE loop; since each of these S_i 's had the property that $S_i \preceq W$, by Theorem 2.6 $S \preceq W$.

Hence we see that it is possible for the function DIRECTED_SEARCH to return a formula S which has the property that $S \preceq W$.

Finally, since $S \preceq W$, therefore by definition $S \rightarrow W$, and hence

$$(H_1 \wedge S) \rightarrow (H_1 \wedge W) \rightarrow H_2,$$

$$\text{i.e. } H_1 \wedge S \rightarrow H_2$$

and thus the last condition for exit from the function is satisfied; therefore GET_APPROX can return S such that $S \preceq W$, and the proof is complete.

Proof of (iii) : Suppose GET_APPROX(H, W) has returned S such that $S \preceq W$ all the n times it has been called, and suppose GET_APPROX is called for the $n + 1^{th}$ time, with first and second arguments H and W respectively. We know that

$$H = B_1 \vee B_2 \vee \dots \vee B_r,$$

where some of the B_i 's constitute the previous approximation for W , and the remaining B_j 's are left-hand sides of verification conditions whose right-hand sides are W . For the B_i 's which constitute the previous approximation for W , we know that each of these B_i 's imply W (since GET-APPROX returned S such that $S \preceq W$ all the n times it has been called so far, therefore any approximation S for W returned by GET-APPROX had the property that $S \rightarrow W$). For the B_j 's which are left-hand sides of verification conditions with W on their right-hand sides, B_j can either be written as H_1 or as $H_1 \wedge W'_k$, for some known formula H_1 and some approximation W'_k to a loop invariant W' (W' could be equal to W). If B_j can be written as H_1 , then $H_1 \rightarrow W$ is valid (since it is a verification condition); if $B_j = H_1 \wedge W'_k$, then since $H_1 \wedge W' \rightarrow W$ is a verification condition, and since GET-APPROX returned S such that $S \preceq W$ all the n times it has been called so far, therefore the approximation W'_k for W' implies W' , therefore we know that

$$\begin{aligned} H_1 \wedge W'_k &\rightarrow H_1 \wedge W' \\ &\rightarrow W \end{aligned}$$

i.e. $H_1 \wedge W'_k \rightarrow W$ is also valid. Hence each B_j implies W .

Therefore $H \rightarrow W$ (since $H = B_1 \vee \dots \vee B_r$ and since $B_k \rightarrow W$ for all $1 \leq k \leq r$).

Proof of (iv) : We prove that GET-APPROX can always return a formula S such that $S \preceq W$ by induction on the number of times GET-APPROX has been called.

Base case : If GET-APPROX has never been called, and it is being called for the first time with first and second arguments H and W respectively, then by (i), $H \rightarrow W$. Therefore by (ii), GET-APPROX can return S such that $S \preceq W$.

Inductive hypothesis : Suppose that for all the n times that GET-APPROX has been called, it has returned formulas S such that $S \preceq W$, where W was the second argument of GET-APPROX in that call.

Inductive step : Suppose GET-APPROX(H, W) is called, this being the $n+1^{th}$ call of GET-APPROX. By the inductive hypothesis, GET-APPROX returned formulas S such that $S \preceq W$ all the n times that GET-APPROX was called (W being the second argument of GET-APPROX in each case). Therefore by (iii), $H \rightarrow W$; and therefore by (ii), GET-APPROX can return S such that $S \preceq W$ in this $n+1^{th}$ call of GET-APPROX too, and the proof is complete by induction.

Proof of (v) : We saw from (iv) that for each loop invariant W^i , it is possible to derive approximations W_j^i (by calling GET-APPROX) such that each W_j^i derived has the property that

$$W_j^i \preceq W^i \quad \forall k.$$

And since $\{F \mid F \preceq W^i\}$ is finite up to variants for any W^i (provided that in the conjunctive normal form of F , no two disjunctions of F are identical, and no

disjunction contains more than one occurrence of any literal) from Theorem 2.5, only a finite number of distinct W_j^i 's exist. But then this means that at some point during the execution of the algorithm, the W_j^i 's derived will start repeating themselves.

Thus there exists some integer λ such that

$$j \geq \lambda \rightarrow (\exists k < \lambda \text{ such that } W_j^i = W_k^i) \text{ for every } i, 1 \leq i \leq n.$$

Recall that $W_j^i \rightarrow W_{j+1}^i$ for every $j \geq 0$. We have to show that there will be a time when $\bigwedge_{i=1}^n \text{flag}(W^i)$ will be true. We first show that $W_\lambda^i = W_{\lambda-1}^i$ for all i , $1 \leq i \leq n$.

We know from the above that there exists $k < \lambda$ such that

$$W_\lambda^i = W_k^i.$$

Therefore $W_\lambda^i = W_k^i \rightarrow W_{k+1}^i \rightarrow W_{k+2}^i \rightarrow \dots \rightarrow W_{\lambda-1}^i \rightarrow W_\lambda^i$

i.e. $W_\lambda^i \rightarrow W_{\lambda-1}^i, W_{\lambda-1}^i \rightarrow W_\lambda^i$.

Therefore $W_{\lambda-1}^i \equiv W_\lambda^i$.

This is true for every i , $1 \leq i \leq n$. Therefore $\text{flag}(W^i)$ will be set to "true" after W_λ^i has been calculated. Since this is true for every i , $1 \leq i \leq n$, $\bigwedge_{i=1}^n \text{flag}(W^i)$ will be true after W_λ^i has been calculated for every i , $1 \leq i \leq n$, and then the algorithm halts.

Note. This claim of completeness does not say that no matter which formulas GET-APPROX generates, the algorithm will terminate with the correct answer. Since the loop invariants W are unknown, there is no way of verifying that an approximation S generated by GET-APPROX indeed satisfies $S \preceq W$; and a potentially infinite number of formulas can be generated by GET-APPROX, only a finite number of which satisfy this condition. However, it does say that if a loop invariant exists, there is a way of deriving it using this algorithm. In the same way, the completeness of resolution does not guarantee that no matter which clauses are chosen to be used in resolution steps, the proof will terminate; rather, it says that there is a way of obtaining a proof, if one exists, if the proper clauses are chosen for resolution. •

3.9 Proof of soundness of the iteration algorithm

To show that this algorithm is sound, all we need to do is to show that the final approximations for the loop invariants which are generated are valid loop invariants. This can be done by showing that all the verification conditions still hold when the generated loop invariants are substituted for the actual loop invariants. Now, when the algorithm terminates, all the flags (for every loop invariant) are set to true. This means that every verification condition with a loop invariant on the right-hand side of the implication sign is true if W_{approx}^i is substituted for the loop invariant in all

such verification conditions. Therefore the only verification conditions which need to be checked are those which do not have a loop invariant on the right-hand side of the implication sign, i.e. those of type (iii) :

$$H_1 \wedge W^i \rightarrow H_2.$$

However, note that for any loop invariant W^i for which a verification condition of the above form exists, the function DIRECTED_SEARCH returns an approximation S for the loop invariant W^i such that $H_1 \wedge S \rightarrow H_2$, since this condition is specifically checked for at the end of the function. Thus the function GET-APPROX also returns an approximation S for the loop invariant W^i such that $H_1 \wedge S \rightarrow H_2$. This means that every approximation W_j^i for W^i satisfies the formula

$$H_1 \wedge W_j^i \rightarrow H_2.$$

Therefore in particular,

$$H_1 \wedge W_{approx}^i \rightarrow H_2$$

and the soundness of the algorithm is proved. •

3.10 A refinement

In this section, we describe a refinement which can improve the efficiency of generating loop invariants. The refinement arises from the fact that assertions which do not mention the program variables need not be included in loop invariants, since they can be generated from the axioms of the programming language operations.

More formally, suppose the cutpoints in a program are numbered $0, 1, 2, \dots, n$, where cutpoints 0 and n are the cutpoints to which the input and output assertions are attached, respectively. Let the inductive assertion attached to cutpoint i be denoted by

$$A_i \wedge B_i$$

where A_i consists of formulas which mention the program variables and B_i consists of formulas which do not mention the program variables. Using the notation from Section 3.2, recall that the verification condition for a path α leading from cutpoint i to cutpoint j is given by :

$$\forall \bar{x} \forall \bar{y} (A_i(\bar{x}, \bar{y}) \wedge B_i \wedge R_\alpha(\bar{x}, \bar{y}) \rightarrow A_j(r_\alpha(\bar{x}, \bar{y})) \wedge B_j)$$

(since B_i and B_j do not mention variables from \bar{x} and \bar{y}). This is equivalent to

$$\begin{aligned} & \forall \bar{x} \forall \bar{y} (A_i(\bar{x}, \bar{y}) \wedge B_i \wedge R_\alpha(\bar{x}, \bar{y}) \rightarrow A_j(r_\alpha(\bar{x}, \bar{y}))) \wedge \\ & \forall \bar{x} \forall \bar{y} (A_i(\bar{x}, \bar{y}) \wedge B_i \wedge R_\alpha(\bar{x}, \bar{y}) \rightarrow B_j) \end{aligned}$$

Since B_j does not mention the program variables,

$$\forall \bar{x} \forall \bar{y} (A_i(\bar{x}, \bar{y}) \wedge B_i \wedge R_\alpha(\bar{x}, \bar{y}) \rightarrow B_j)$$

is equivalent to

$$\forall \bar{x} \forall \bar{y} (A_i(\bar{x}, \bar{y}) \wedge B_i \rightarrow B_j)$$

We assume that this verification condition can be proved from the set of axioms “*AXIOMS*” for the programming language operations; thus we have

$$\forall \bar{x} \forall \bar{y} (AXIOMS \wedge A_i(\bar{x}, \bar{y}) \wedge B_i \rightarrow B_j).$$

This is equivalent to

$$AXIOMS \wedge (\exists \bar{x} \exists \bar{y}) (A_i(\bar{x}, \bar{y}) \wedge B_i \rightarrow B_j)$$

(since none of \bar{x} , \bar{y} occur free in B_i and B_j). It is reasonable to assume that

$$AXIOMS \rightarrow (\exists \bar{x} \exists \bar{y}) (A_i(\bar{x}, \bar{y})).$$

Thus we have

$$AXIOMS \wedge B_i \rightarrow B_j.$$

Now, clearly $AXIOMS \rightarrow B_0$ (recall that $A_0 \wedge B_0$ is the input assertion). Also, from the above, $AXIOMS \wedge B_0 \rightarrow B_j$ is valid for all j such that the cutpoint j is reachable from the cutpoint 0 by a path without intervening assertions. Thus $AXIOMS \rightarrow B_j$ for all such j . But then by the connectedness of the program, we can inductively extend this argument to show that $AXIOMS \rightarrow B_j$ for all j such that $0 \leq j \leq n$. Thus all the B_j 's can be dropped from the inductive assertions.

The above result makes the task of generating loop invariants more efficient, since any formula which is generated by the function GET-APPROX and which does not contain program variables can be immediately discarded. This greatly reduces the search space for a valid loop invariant.

3.11 Some examples

Example 3.1 The program over the integers shown in Figure 3.2 computes $z = \text{gcd}(x_1, x_2)$ for every pair of positive integers x_1 and x_2 ; that is, z is the greatest common divisor of x_1 and x_2 . The computation method is based on the fact that

$$\text{If } y_1 > y_2, \text{ then } \text{gcd}(y_1, y_2) = \text{gcd}(y_1 - y_2, y_2)$$

$$\text{If } y_1 < y_2, \text{ then } \text{gcd}(y_1, y_2) = \text{gcd}(y_1, y_2 - y_1)$$

$$\text{If } y_1 = y_2, \text{ then } \text{gcd}(y_1, y_2) = y_1 = y_2.$$

The program is to be proved partially correct with respect to the input predicate $\phi(\bar{x}) : x_1 > 0 \wedge x_2 > 0$ and the output predicate $\psi(\bar{x}, z) : z = \text{gcd}(x_1, x_2)$. The two loops of the program have been cut at point B and an unknown loop invariant W^1 attached to this point. We will use the iteration algorithm to derive this invariant.

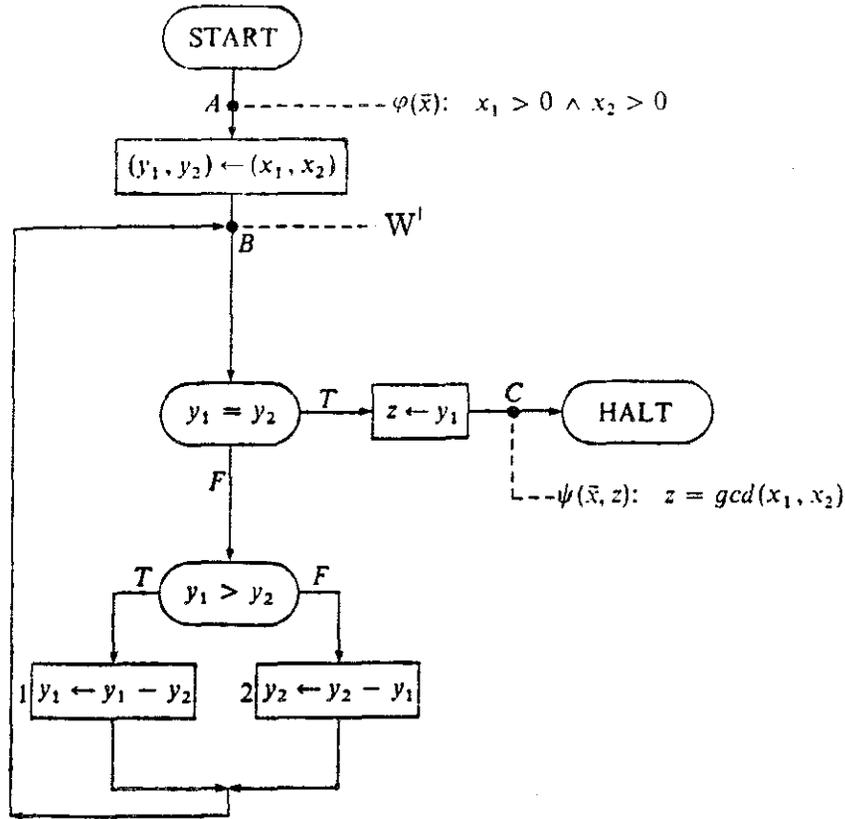


Figure 3.2 Calculating the g.c.d. of two numbers

Since the domain of the given program is the set of integers, and the operations of the language include arithmetic operations, comparison and equality, we must include the necessary axioms for arithmetic operations, comparison and equality when performing resolutions. Also, we must provide the definition of the *gcd* function, since the function is mentioned in the output assertion ψ . The axioms listed above are used to define the *gcd* function. Let the set of all these axioms be *AXIOMS*.

We perform the iteration algorithm step by step. There is only one loop invariant here, therefore $n = 1$. There are four paths leading from one cutpoint to another, since two different paths exist in the program loop, depending on which branch is taken after the test $y_1 > y_2$. We denote old values for the variables y_1 and y_2 by y_1' and y_2' respectively.

There are four verification conditions for the program, which are
 $vc_1 \equiv (x_1 > 0) \wedge (x_2 > 0) \wedge (x_1 = y_1) \wedge (x_2 = y_2) \rightarrow W^1(\bar{x}, y_1, y_2)$

$$\begin{aligned}
vc_2 &\equiv \exists y'_1 (W^1(\bar{x}, y'_1, y_2) \wedge (y_1 = y'_1 - y_2) \wedge (y'_1 \neq y_2) \wedge (y'_1 > y_2) \rightarrow W^1(\bar{x}, y_1, y_2)) \\
vc_3 &\equiv \exists y'_2 (W^1(\bar{x}, y_1, y'_2) \wedge (y_2 = y'_2 - y_1) \wedge (y_1 \neq y'_2) \wedge (y_1 \leq y'_2) \rightarrow W^1(\bar{x}, y_1, y_2)) \\
vc_4 &\equiv W^1(\bar{x}, y_1, y_2) \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2)).
\end{aligned}$$

Step 1. $index(W^1) = 0$, $W_0^1 = false$

Step 2. $list = [W^1]$.

Step 3. First iteration :

$$W = W^1$$

$$T = \{lhs(vc_1)\}$$

$$index(W^1) = 1$$

$$flag(W^1) = false$$

$$W_1^1 = GET-APPROX(W_0^1 \vee lhs(vc_1), W^1)$$

$$= GET-APPROX(false \vee lhs(vc_1), W^1)$$

$$= GET-APPROX(lhs(vc_1), W^1)$$

$$= GET-APPROX(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2, W^1).$$

Call GET-APPROX(H, W^1), where

$$H = x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2$$

input b to be some large number

$$S := DIRECTED_SEARCH(H, W^1)$$

Call DIRECTED_SEARCH(H, W^1)

check if $AXIOMS \wedge H \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$; since this is valid, continue;

$$S := true;$$

$$H = B_1;$$

WHILE LOOP :

$$(r = 1)$$

$$num.iterations := 1$$

First FOR loop :

PROOFS := set of all clauses generated from $B_1 \wedge AXIOMS$ by resolution during some proof of $AXIOMS \wedge H \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of $AXIOMS \wedge H \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$ is given at the end of this example; thus we have

PROOFS := $\{\{x_1 > 0\}, \{x_2 > 0\}, \{x_1 = y_1\}, \{x_2 = y_2\}, \{y_1 \neq y_2, y_1 = gcd(y_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, y_2)\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, x_2)\}\}$;

From this set, choose $S_1 =$ all clauses in PROOFS, leaving out axioms $= x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$;

Second FOR loop :

Since $B_1 \rightarrow S_1$, therefore $S := (true \wedge S_1) =$

$$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2))) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)).$$

Clearly $AXIOMS \wedge S \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$ is valid, therefore DIRECTED_SEARCH and GET-APPROX return $(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2))) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$.

Hence we obtained, after calling the function GET-APPROX,

$$W_1^1 \equiv (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2))) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)).$$

Second iteration of Step 3 :

$$W = W^1$$

$$T = \{lhs(vc_2), lhs(vc_3)\}$$

$$index(W^1) = 2$$

$$flag(W^1) = false$$

$$W_2^1 = GET-APPROX(W_1^1 \vee lhs(vc_2) \vee lhs(vc_3), W^1)$$

Call GET-APPROX(H, W^1), where

$$H = (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2))) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)) \vee$$

$$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1' \wedge x_2 = y_2 \wedge (y_1 = y_1' - y_2) \wedge (y_1' \neq y_2) \wedge y_1' > y_2) \vee$$

$$(x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2' \wedge (y_2 = y_2' - y_1) \wedge (y_2' \neq y_1) \wedge y_1 \leq y_2')$$

$$= B_1 \vee B_2 \vee B_3.$$

input b to be some large number

$$S := DIRECTED_SEARCH(H, W^1)$$

Call DIRECTED_SEARCH(H, W^1)

check if $AXIOMS \wedge H \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$; since this is valid, continue;

$$S := true;$$

$$H = B_1 \vee B_2 \vee B_3$$

First execution of WHILE loop :

$$(r = 3)$$

$$num_iterations := 1$$

First execution of first FOR loop : $i = 1$:

PROOFS := set of all clauses generated from $B_1 \wedge AXIOMS$ by resolution during some proof of $AXIOMS \wedge B_1 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of $AXIOMS \wedge B_1 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$ is given at the end of this example; thus we have

PROOFS := $\{\{x_1 > 0\}, \{x_2 > 0\}, \{x_1 = y_1\}, \{x_2 = y_2\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{y_1 \neq y_2, y_1 = gcd(y_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, y_2)\}, \{y_1 \neq y_2, y_1 = gcd(x_1, x_2)\}\}$;

From this set, choose $S_1 =$ all clauses in PROOFS, leaving out axioms $= x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2))$;

Second execution of first FOR loop : $i = 2$:

PROOFS := set of all clauses generated from $B_2 \wedge AXIOMS$ by resolution during some proof of $AXIOMS \wedge B_2 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of $AXIOMS \wedge B_2 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$ is given at the end of this example; thus we have

PROOFS := $\{\{\neg(Y > Z), gcd(Y, Z) = gcd(Y - Z, Z)\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{x_1 > 0\}, \{x_2 > 0\}, \{y'_1 > y_2\}, \{x_1 = y'_1\}, \{x_2 = y_2\}, \{y_1 = y'_1 - y_2\}, \{x_1 > y_2\}, \{x_1 > x_2\}, \{gcd(x_1, x_2) = gcd(x_1 - x_2, x_2)\}, \{y_1 = x_1 - y_2\}, \{y_1 = x_1 - x_2\}, \{gcd(x_1, x_2) = gcd(y_1, x_2)\}, \{gcd(x_1, x_2) = gcd(y_1, y_2)\}\}$;

From this set, choose $S_2 =$ all clauses generated from $B_2 \wedge AXIOMS$ in PROOFS, leaving out axioms

$= (x_1 > 0 \wedge x_2 > 0 \wedge y'_1 > y_2 \wedge x_1 = y'_1 \wedge x_2 = y_2 \wedge y_1 = y'_1 - y_2 \wedge x_1 > y_2 \wedge x_1 > x_2 \wedge (gcd(x_1, x_2) = gcd(x_1 - x_2, x_2)) \wedge y_1 = x_1 - y_2 \wedge y_1 = x_1 - x_2 \wedge (gcd(x_1, x_2) = gcd(y_1, x_2)) \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$.

Third execution of first FOR loop : $i = 3$:

PROOFS := set of all clauses generated from $B_3 \wedge AXIOMS$ by resolution during some proof of $AXIOMS \wedge B_3 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$.

A proof of $AXIOMS \wedge B_3 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$ is given at the end of this example; thus we have

PROOFS := $\{\{\neg(Y < Z), gcd(Y, Z) = gcd(Y, Z - Y)\}, \{Y \neq Z, Y = gcd(Y, Z)\}, \{x_1 > 0\}, \{x_2 > 0\}, \{y_1 < y'_2, y_1 = y'_2\}, \{x_2 = y'_2\}, \{x_1 = y_1\}, \{y_2 = y'_2 - y_1\}, \{y'_2 \neq y_1\}, \{y_1 < y'_2\}, \{y_1 < x_2\}, \{x_1 < x_2\}, \{gcd(x_1, x_2) = gcd(x_1, x_2 - x_1)\}, \{y_2 = x_2 - y_1\}, \{y_2 = x_2 - x_1\}, \{gcd(x_1, x_2) = gcd(x_1, y_2)\}, \{gcd(x_1, x_2) = gcd(y_1, y_2)\}\}$;

From this set, choose $S_3 =$ all clauses generated from $B_3 \wedge AXIOMS$ in PROOFS, leaving out axioms

$= (x_1 > 0 \wedge x_2 > 0 \wedge (y_1 < y'_2 \vee y_1 = y'_2) \wedge x_2 = y'_2 \wedge x_1 = y_1 \wedge y_2 =$

$$y'_2 - y_1 \wedge y'_2 \neq y_1 \wedge y_1 < y'_2 \wedge y_1 < x_2 \wedge x_1 < x_2 \wedge (\gcd(x_1, x_2) = \gcd(x_1, x_2 - x_1)) \wedge y_2 = x_2 - y_1 \wedge y_2 = x_2 - x_1 \wedge (\gcd(x_1, x_2) = \gcd(x_1, y_2)) \wedge \gcd(x_1, x_2) = \gcd(y_1, y_2)).$$

Second FOR loop :

$i = 1 :$

We find that

$$\begin{aligned} B_1 \rightarrow S_1 & \text{ is valid;} \\ B_2 \rightarrow S_1 & \text{ is not valid;} \\ B_3 \rightarrow S_1 & \text{ is not valid.} \end{aligned}$$

$i = 2 :$

We find that

$$\begin{aligned} B_1 \rightarrow S_2 & \text{ is not valid;} \\ B_2 \rightarrow S_2 & \text{ is valid;} \\ B_3 \rightarrow S_2 & \text{ is not valid.} \end{aligned}$$

$i = 3 :$

We find that

$$\begin{aligned} B_1 \rightarrow S_3 & \text{ is not valid;} \\ B_2 \rightarrow S_3 & \text{ is not valid;} \\ B_3 \rightarrow S_3 & \text{ is valid.} \end{aligned}$$

Since $S = true$, the WHILE loop must be repeated again.

The WHILE loop is now repeated several times, with S_i chosen to be a different subset of PROOFS each time (for $i = 1, 2, 3$). After a number of iterations, we finally choose

$$S_2 = x_1 > 0 \wedge x_2 > 0 \wedge \gcd(x_1, x_2) = \gcd(y_1, y_2),$$

which satisfies

$$\begin{aligned} B_1 \rightarrow S_2 \\ B_2 \rightarrow S_2 \\ B_3 \rightarrow S_2. \end{aligned}$$

Thus S is set to be

$$S = x_1 > 0 \wedge x_2 > 0 \wedge \gcd(x_1, x_2) = \gcd(y_1, y_2).$$

Clearly $AXIOMS \wedge S \wedge (y_1 = y_2) \rightarrow (y_1 = \gcd(x_1, x_2))$ is valid, therefore DIRECTED_SEARCH and GET-APPROX return $(x_1 > 0 \wedge x_2 > 0 \wedge \gcd(x_1, x_2) = \gcd(y_1, y_2))$

Hence we obtained, after calling the function GET-APPROX,

$$W_2^1 = (x_1 > 0 \wedge x_2 > 0 \wedge \gcd(x_1, x_2) = \gcd(y_1, y_2))$$

Third iteration of Step 3 :

$$W = W^1$$

$T = \emptyset$ (it can be verified that all the verification conditions are valid with W_2^1 substituted for W^1 everywhere)

$$index(W^1) = 3$$

$$flag(W^1) = true$$

$$W_3^1 = W_2^1$$

Since $flag(W^1)$ is true, Step 3 terminates.

4. $W_{approx}^1 = (x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2))$

5. Halt.

The loop invariant derived is

$$W^1 = (x_1 > 0 \wedge x_2 > 0 \wedge gcd(x_1, x_2) = gcd(y_1, y_2)).$$

The proofs used in this example were :

Proof of $AXIOMS \wedge x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$:

(The same proof can be used to show that $AXIOMS \wedge x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2 \wedge (y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)) \wedge (y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)) \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$)

1. $x_1 > 0$ Given
2. $x_2 > 0$ Given
3. $x_1 = y_1$ Given
4. $x_2 = y_2$ Given
5. $Y \neq Z \vee Y = gcd(Y, Z)$ Axiom
6. $(y_1 = y_2)$ Given
7. $(y_1 \neq gcd(x_1, x_2))$ Given
8. $y_1 \neq y_2 \vee y_1 = gcd(y_1, y_2)$ Instance of 5
9. $y_1 \neq y_2 \vee y_1 = gcd(x_1, y_2)$ Paramodulate 3,8
10. $y_1 \neq y_2 \vee y_1 = gcd(x_1, x_2)$ Paramodulate 4,9
11. $y_1 = gcd(x_1, x_2)$ Resolve 6,10
12. empty clause Resolve 7,11.

Proof of $AXIOMS \wedge (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1' \wedge x_2 = y_2 \wedge (y_1 = y_1' - y_2) \wedge (y_1' \neq y_2) \wedge y_1' > y_2 \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2))$:

1. $\neg(Y > Z) \vee gcd(Y, Z) = gcd(Y - Z, Z)$ Axiom
2. $Y \neq Z \vee Y = gcd(Y, Z)$ Axiom
3. $y_1' > y_2$ Given
4. $x_1 = y_1'$ Given
5. $x_2 = y_2$ Given
6. $y_1 = y_1' - y_2$ Given

7. $y_1 = y_2$ Given
8. $y_1 \neq gcd(x_1, x_2)$ Given
9. $x_1 > y_2$ Paramodulate 3,4
10. $x_1 > x_2$ Paramodulate 5,9
11. $gcd(x_1, x_2) = gcd(x_1 - x_2, x_2)$ Resolve 1,10
12. $y_1 = x_1 - y_2$ Paramodulate 6,4
13. $y_1 = x_1 - x_2$ Paramodulate 5,12
14. $gcd(x_1, x_2) = gcd(y_1, x_2)$ Paramodulate 11,13
15. $gcd(x_1, x_2) = gcd(y_1, y_2)$ Paramodulate 5,14
16. $y_1 = gcd(y_1, y_2)$ Resolve 2,7
17. $gcd(x_1, x_2) = y_1$ Paramodulate 15,16
18. empty clause Resolve 8,17

Proof of $AXIOMS \wedge (x_1 > 0 \wedge x_2 > 0 \wedge x_1 = y_1 \wedge x_2 = y_2' \wedge (y_2 = y_2' - y_1) \wedge (y_2' \neq y_1) \wedge y_1 \leq y_2' \wedge (y_1 = y_2) \rightarrow (y_1 = gcd(x_1, x_2)))$:

1. $\neg(Y < Z) \vee gcd(Y, Z) = gcd(Y, Z - Y)$ Axiom
2. $Y \neq Z \vee Y = gcd(Y, Z)$ Axiom
3. $y_1 < y_2' \vee y_1 = y_2'$ Given
4. $x_2 = y_2'$ Given
5. $x_1 = y_1$ Given
6. $y_2 = y_2' - y_1$ Given
7. $y_2' \neq y_1$ Given
8. $y_1 = y_2$ Given
9. $y_1 \neq gcd(x_1, x_2)$ Given
10. $y_1 < y_2'$ Resolve 3,7
11. $y_1 < x_2$ Paramodulate 4,10
12. $x_1 < x_2$ Paramodulate 5,11
13. $gcd(x_1, x_2) = gcd(x_1, x_2 - x_1)$ Resolve 1,12
14. $y_2 = x_2 - y_1$ Paramodulate 6,4
15. $y_2 = x_2 - x_1$ Paramodulate 5,14
16. $gcd(x_1, x_2) = gcd(x_1, y_2)$ Paramodulate 13,15
17. $gcd(x_1, x_2) = gcd(y_1, y_2)$ Paramodulate 5,16
18. $y_1 = gcd(y_1, y_2)$ Resolve 2,8
19. $gcd(x_1, x_2) = y_1$ Paramodulate 17,18
20. empty clause Resolve 9,19 •

Example 3.2 This example illustrates the working of the iteration algorithm when unskolemization is used to derive the loop invariant. The flowchart program in Figure 3.3 multiplies two numbers by repeated addition of one number to a variable.

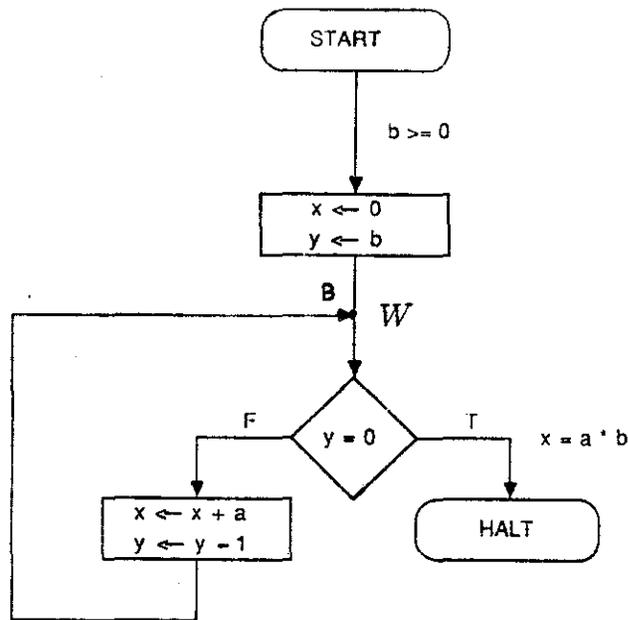


Figure 3.3 Multiplying two numbers by repeated addition

The program is to be proved partially correct with respect to the input predicate $\phi(x, y) : b \geq 0$ and the output predicate $\psi(x) : x = a * b$. The loop of the program has been cut at point B and an unknown loop invariant W attached to this point. We will use the iteration algorithm to derive this invariant.

Since the domain of the given program is the set of integers, and the operations of the language include arithmetic operations and equality, we must include the necessary axioms for arithmetic operations and equality when performing resolutions. Let the set of all these axioms be *AXIOMS*.

We perform the iteration algorithm step by step. There is only one loop invariant here, therefore $n = 1$. There are three paths leading from one cutpoint to another. We denote old values for the variables x and y by x' and y' respectively. As in the previous example, proofs of unsatisfiability obtained in the function *DIRECTED_SEARCH* can be found at the end of this example. The three verification conditions for this program are :

1. $(y = b) \wedge (b \geq 0) \wedge (x = 0) \rightarrow W(x, y)$
2. $\exists x' \exists y' (W(x', y') \wedge y' \neq 0 \wedge (x = x' + a) \wedge (y = y' - 1) \rightarrow W(x, y))$
3. $W(x, y) \wedge (y = 0) \rightarrow (x = a * b)$.

Tracing the iteration algorithm, we get

1. $index(W) = 0$

2. $list = [W]$

3. First iteration of Step 3 :

$W_0 = false$

$T = \{lhs(vc_1)\}$

$flag(W) = false$

$W_1 = GET-APPROX(W_0 \vee lhs(vc_1), W)$

$= GET-APPROX(false \vee ((y = b) \wedge (b \geq 0) \wedge (x = 0)), W)$

$= GET-APPROX(((y = b) \wedge (b \geq 0) \wedge (x = 0)), W)$

$H = B_1 = ((y = b) \wedge (b \geq 0) \wedge (x = 0))$

call DIRECTED_SEARCH(((y = b) \wedge (b \geq 0) \wedge (x = 0)), W)

PROOFS = $\{\{x = 0\}, \{y = b\}, \{b \geq 0\}, \{Z * 0 = 0\}\}$;

Choose $S_1 =$ all clauses in PROOFS except axioms

$= ((y = b) \wedge (x = 0) \wedge (b \geq 0))$;

return $S = ((y = b) \wedge (b \geq 0) \wedge (x = 0))$

Second iteration of Step 3 :

$W_1 = ((y = b) \wedge (b \geq 0) \wedge (x = 0))$

$T = \{lhs(vc_2)\}$

$flag(W) = false$

$W_2 = GET-APPROX(W_1 \vee lhs(vc_2), W)$

$= GET-APPROX(((y = b) \wedge (b \geq 0) \wedge (x = 0))$

$\vee ((y' = b) \wedge (b \geq 0) \wedge (x' = 0) \wedge y' \neq 0 \wedge (x = x' + a) \wedge (y = y' - 1)), W)$

$H = B_1 \vee B_2 = ((y = b) \wedge (b \geq 0) \wedge (x = 0))$

$\vee ((y' = b) \wedge (b \geq 0) \wedge (x' = 0) \wedge y' \neq 0 \wedge (x = x' + a) \wedge (y = y' - 1))$

call DIRECTED_SEARCH($B_1 \vee B_2, W$)

First execution of first FOR loop :

PROOFS = $\{\{x = 0\}, \{y = b\}, \{b \geq 0\}, \{Z * 0 = 0\}\}$;

Choose $S_1 =$ all clauses in PROOFS except axioms

$= ((y = b) \wedge (x = 0) \wedge (b \geq 0))$;

Second execution of first FOR loop :

PROOFS = $\{\{x = a\}, \{y = b - 1\}, \{b \geq 0\}, \{y' = b\}, \{y = y' - 1\}, \{x' = 0\}, \{x = x' + a\}, \{x = 0 + a\}, \{x = a + 0\}, \{Y * Z = Z * Y\}, \{Z * 1 = Z\}, \{X - Y \neq Z, X = Y + Z\}, \{Z = Z + 0\}\}$;

Choose $S_2 =$ all clauses in PROOFS except axioms

$= ((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0))$;

Second FOR loop :

$i = 1$; we find that

$B_1 \rightarrow S_1$ is valid;

$B_2 \rightarrow S_1$ is not valid;

$i = 2$; we find that

$B_1 \rightarrow S_2$ is not valid;

$B_2 \rightarrow S_2$ is valid;

Since $S = true$, the WHILE loop must be repeated again. After several repetitions, however, no formula is derived such that B_1 and B_2 both imply it. Thus we exit from the WHILE loop.

We get $S = S_1 \vee S_2$

$$\begin{aligned} &= ((y = b) \wedge (x = 0) \wedge (b \geq 0)) \vee \\ &((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0)); \\ \text{return } S &= ((y = b) \wedge (x = 0) \wedge (b \geq 0)) \vee ((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0)) \end{aligned}$$

Third iteration of Step 3 :

$$W_2 = ((y = b) \wedge (b \geq 0) \wedge (x = 0)) \vee ((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0))$$

$$T = \{lhs(vc_2)\}$$

$$flag(W) = false$$

$$W_3 = \text{GET-APPROX}(W_2 \vee lhs(vc_2), W)$$

$$= \text{GET-APPROX}(((y = b) \wedge (b \geq 0) \wedge (x = 0))$$

$$\vee ((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0))$$

$$\vee ((y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge$$

$$(((y' = b) \wedge (b \geq 0) \wedge (x' = 0)) \vee ((y' = b - 1) \wedge (x' = a) \wedge (b \geq 0) \wedge (y'' = b) \wedge (y' = y'' - 1) \wedge (x'' = 0) \wedge (x' = x'' + a) \wedge (x' = 0 + a) \wedge (x' = a + 0))), W)$$

$$H = ((y = b) \wedge (b \geq 0) \wedge (x = 0))$$

$$\vee ((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0))$$

$$\vee ((y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y' = b) \wedge (b \geq 0) \wedge (x' = 0))$$

$$\vee ((y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y' = b - 1) \wedge (x' = a) \wedge (b \geq 0) \wedge (y'' = b) \wedge (y' = y'' - 1) \wedge (x'' = 0) \wedge (x' = x'' + a) \wedge (x' = 0 + a) \wedge (x' = a + 0))$$

$$\equiv ((y = b) \wedge (b \geq 0) \wedge (x = 0))$$

$$\vee ((y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y' = b) \wedge (b \geq 0) \wedge (x' = 0))$$

$$\vee ((y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y' = b - 1) \wedge (x' = a) \wedge (b \geq 0) \wedge (y'' = b) \wedge (y' = y'' - 1) \wedge (x'' = 0) \wedge (x' = x'' + a) \wedge (x' = 0 + a) \wedge (x' = a + 0))$$

$$= B_1 \vee B_2 \vee B_3$$

call DIRECTED_SEARCH($B_1 \vee B_2 \vee B_3, W$)

First execution of first FOR loop :

PROOFS = $\{\{x = 0\}, \{y = b\}, \{b \geq 0\}, \{Z * 0 = 0\}\}$;

Choose S_1 = all clauses in PROOFS except axioms

= $((y = b) \wedge (x = 0) \wedge (b \geq 0))$;

Second execution of first FOR loop :

PROOFS = $\{\{x = a\}, \{y = b - 1\}, \{b \geq 0\}, \{y' = b\}, \{y = y' - 1\}, \{x' = 0\}, \{x = x' + a\}, \{x = 0 + a\}, \{x = a + 0\}, \{Y * Z = Z * Y\}, \{Z * 1 = Z\}, \{X - Y \neq Z, X = Y + Z\}, \{Z + 0 = Z\}\}$;

Choose S_2 = all clauses in PROOFS except axioms

= $((y = b - 1) \wedge (x = a) \wedge (b \geq 0) \wedge (y' = b) \wedge (y = y' - 1) \wedge (x' = 0) \wedge (x = x' + a) \wedge (x = 0 + a) \wedge (x = a + 0))$;

Third execution of first FOR loop :

PROOFS = $\{\{x = a * 2\}, \{y = b - 2\}, \{b \geq 0\}, \{y' = b - 1\}, \{x' = a\}, \{y = y' - 1\}, \{x = x' + a\}, \{y = b - 1 - 1\}, \{x = a + a\}, \{Z * 2 = Z + Z\}, \{-1 - 1 = -2\}, \{X - Y \neq Z, X = Y + Z\}, \{Z = Z + 0\}, \{Y * Z = Z * Y\}\}$;

Choose S_3 = *unsk*(all clauses in PROOFS except axioms)

= *unsk* $((x = a * 2) \wedge (y = b - 2) \wedge (b \geq 0) \wedge (y' = b - 1) \wedge (x' = a) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y = b - 1 - 1) \wedge (x = a + a))$;

= $\exists z((y = b - z) \wedge (x = a * z) \wedge (b \geq 0) \wedge (y' = b - 1) \wedge (x' = a) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y = b - 1 - 1) \wedge (x = a + a))$;

(obtained by unskolemizing the symbol "2" and replacing it by the existentially quantified variable z)

Second FOR loop :

$i = 1$:

We find that

$B_1 \rightarrow S_1$ is valid;

$B_2 \rightarrow S_1$ is not valid;

$B_3 \rightarrow S_1$ is not valid.

$i = 2$:

We find that

$B_1 \rightarrow S_2$ is not valid;

$B_2 \rightarrow S_2$ is valid;

$B_3 \rightarrow S_2$ is not valid.

$i = 3$:

We find that

$B_1 \rightarrow S_3$ is not valid;

$B_2 \rightarrow S_3$ is not valid;

$B_3 \rightarrow S_3$ is valid.

Since $S = true$, the WHILE loop must be repeated again.

The WHILE loop is now repeated several times, with S_i chosen to be a different subset of PROOFS each time (for $i = 1, 2, 3$). After a number of iterations, we finally choose

$$S_3 = \exists z((y = b - z) \wedge (x = a * z) \wedge (b \geq 0)),$$

which satisfies

$$B_1 \rightarrow S_2$$

$$B_2 \rightarrow S_2$$

$$B_3 \rightarrow S_2.$$

Thus S is set to be

$$S = \exists z((y = b - z) \wedge (x = a * z) \wedge (b \geq 0))$$

$$\text{return } S = \exists z((y = b - z) \wedge (x = a * z) \wedge (b \geq 0))$$

Fourth iteration of Step 3 :

$T = \emptyset$ (it can be verified that all the verification conditions are valid with W_3 substituted for W everywhere)

$$flag(W) = true$$

$$W_4 = W_3$$

Since $flag(W)$ is true, Step 3 terminates.

4. $W_{approx} = \exists z(x = a * z \wedge y = b - z) \wedge b \geq 0$

5. Halt.

The proofs used in this example were :

Proof of $AXIOMS \wedge x = 0 \wedge y = b \wedge y = 0 \rightarrow x = a * b$:

1. $x = 0$ Given

2. $y = b$ Given

3. $Z * 0 = 0$ Axiom

4. $y = 0$ Given

5. $x \neq a * b$ Given

6. $0 \neq a * b$ Paramodulate 1,5

7. $0 \neq a * y$ Paramodulate 2,6

8. $0 \neq a * 0$ Paramodulate 4,7

9. empty clause Resolve 3,8.

Proof of $AXIOMS \wedge y' = b \wedge b \geq 0 \wedge x' = 0 \wedge y' \neq 0 \wedge x = x' + a \wedge y = y' - 1 \wedge y = 0 \rightarrow x = a * b$:

1. $y' = b$ Given

2. $x' = 0$ Given
3. $x = x' + a$ Given
4. $y = y' - 1$ Given
5. $Z * 1 = Z$ Axiom
6. $X - Y \neq Z, X = Y + Z$ Axiom
7. $Z = Z + 0$ Axiom
8. $Y * Z = Z * Y$ Axiom
9. $y = 0$ Given
10. $x \neq a * b$ Given
11. $y = b - 1$ Paramodulate 1,4
12. $x = 0 + a$ Paramodulate 2,3
13. $x = a + 0$ Paramodulate 8,12
14. $0 = b - 1$ Paramodulate 9,11
15. $b = 1 + 0$ Resolve 6,14
16. $b = 1$ Paramodulate 7,15
17. $x \neq a * 1$ Paramodulate 10,16
18. $x \neq a$ Paramodulate 5,17
19. $x = a$ Paramodulate 7,13
20. empty clause Resolve 18,19.

Proof of $AXIOMS \wedge (y' \neq 0) \wedge (y = y' - 1) \wedge (x = x' + a) \wedge (y' = b - 1) \wedge (x' = a) \wedge (b \geq 0) \wedge (y'' = b) \wedge (y' = y'' - 1) \wedge (x'' = 0) \wedge (x' = x'' + a) \wedge (x' = 0 + a) \wedge (x' = a + 0) \wedge y = 0 \rightarrow x = a * b$:

1. $y' = b - 1$ Given
2. $y = y' - 1$ Given
3. $x' = a$ Given
4. $x = x' + a$ Given
5. $X - Y \neq Z, X = Y + Z$ Axiom
6. $Z * 2 = Z + Z$ Axiom
7. $-1 - 1 = -2$ Axiom
8. $Z = Z + 0$ Axiom
9. $Y * Z = Z * Y$ Axiom
10. $y = 0$ Given
11. $x \neq a * b$ Given
12. $y = b - 1 - 1$ Paramodulate 1,2
13. $x = a + a$ Paramodulate 3,4
14. $x = a * 2$ Paramodulate 6,13
15. $y = b - 2$ Paramodulate 7,12
16. $0 = b - 2$ Paramodulate 10,15

17. $b = 2 + 0$ Resolve 5,16
18. $b = 2$ Paramodulate 8,17
19. $x \neq a * 2$ Paramodulate 11,18
20. empty clause Resolve 14,19.

Comments : The unskolemization which was performed for the symbol “2” in the third iteration of Step 3 could have been performed for the symbols “0” or “1” in the first and second iterations of Step 3 respectively, but this would have required some rewriting; the clause $\{x = 0\}$ would have had to be rewritten as $\{x = a * 0\}$ in the first case, and the clause $\{x = a\}$ would have had to be rewritten as $\{x = a * 1\}$ in the second case. It is not easy to “guess” how this rewriting should be done. However, in the third iteration of Step 3, the replacement of the symbol “2” by an existentially quantified variable was more straightforward, since no imaginative rewriting was required (the rewriting of $a + a$ to $a * 2$ and of $b - 1 - 1$ to $b - 2$ was performed during the last proof shown above since the theorem to be refuted required this rewriting).

Example 3.3 In this example, a loop invariant for the flowchart program shown in Figure 3.1 is derived. In Section 3.1, we showed how a loop invariant for this program was derived using Wegbreit’s top-down approach. The same program was also verified by King [King 69]. The program calculates the quotient and remainder of numbers x and y and is to be proved partially correct with respect to the input predicate $\phi(x, y) : x \geq 0 \wedge y > 0$ and the output predicate $\psi(x) : x = q * y + r \wedge 0 \leq r \wedge r < y$. An unknown loop invariant W for the program loop is to be generated.

Since the domain of the given program is the set of integers, and the operations of the language include arithmetic operations and equality, we must include the necessary axioms for arithmetic operations and equality when performing resolutions. Let the set of all these axioms be *AXIOMS*.

We perform the iteration algorithm step by step. There is only one loop invariant here, therefore $n = 1$. There are three paths leading from one cutpoint to another. We denote new values for the variables q and r by q' and r' respectively. As in the previous example, proofs of unsatisfiability obtained in the function DIRECTED_SEARCH can be found at the end of this example. The three verification conditions for this program are :

1. $x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x \rightarrow W(q, r)$
2. $\exists q' \exists r' (W(q, r) \wedge r \geq y \wedge q' = q + 1 \wedge r' = r - y \rightarrow W(q', r'))$
3. $W(q, r) \wedge r < y \rightarrow x = q * y + r \wedge 0 \leq r \wedge r < y$.

Tracing the iteration algorithm, we get

1. $index(W) = 0$

2. $list = [W]$

3. First iteration of Step 3 :

$W_0 = false$

$T = \{lhs(vc_1)\}$

$flag(W) = false$

$W_1 = GET-APPROX(W_0 \vee lhs(vc_1), W)$

$= GET-APPROX(false \vee (x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x), W)$

$= GET-APPROX((x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x), W)$

$H = B_1 = x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x$

call DIRECTED_SEARCH($x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x, W$)

Prove that $(x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x) \wedge r < y \rightarrow x = q * y + r \wedge 0 \leq r \wedge r < y$

$PROOFS = \{\{x \geq 0\}, \{q = 0\}, \{r = x\}, \{0 * Z = Z\}, \{q * Z = 0\}, \{X \neq Y, Z + X = Z + Y\}, \{Z + r = Z + x\}, \{q * y + r = q * y + x\}, \{q * y + r = 0 + x\}, \{0 + Z = Z\}, \{q * y + r = x\}, \{r \geq 0\}\}$

Choose S_1 to be the last two clauses in the set $PROOFS$ listed above;
i.e.

$S_1 = r \geq 0 \wedge q * y + r = x$

return $S = r \geq 0 \wedge q * y + r = x$.

Second iteration of Step 3 :

$T = \emptyset$ (it can be verified that all the verification conditions are valid with

W_1 substituted for W everywhere)

$flag(W) = true$

$W_2 = W_1$

4. $W_{approx} = r \geq 0 \wedge q * y + r = x$.

5. Halt.

The proof used in this example was :

Proof of $AXIOMS \wedge x \geq 0 \wedge y > 0 \wedge q = 0 \wedge r = x \wedge r < y \rightarrow x = q * y + r \wedge r \geq 0 \wedge r < y$:

1. $x \geq 0$ Given

2. $q = 0$ Given

3. $r = x$ Given

4. $r < y$ Given

5. $x \neq q * y + r \vee \neg(r \geq 0) \vee r \geq y$ Given

6. $0 * Z = 0$ Axiom

7. $q * Z = 0$ Paramodulate 2,6

8. $X \neq Y \vee Z + X = Z + Y$ Axiom
9. $Z + r = Z + x$ Resolve 3,8
10. $q * y + r = q * y + x$ factor of 9
11. $q * y + r = 0 + x$ Paramodulate 7,10
12. $0 + Z = Z$ Axiom
13. $q * y + r = x$ Paramodulate 11,12
14. $r \geq 0$ Paramodulate 1,3
15. $\neg(r \geq 0) \vee r \geq y$ Resolve 5, 13
16. $r \geq y$ Resolve 14,15
17. $\neg(Y \geq Z), \neg(Y < Z)$ Axiom
18. $\neg(r \geq y)$ Resolve 4, 17
19. empty clause Resolve 16, 18.

4. Learning from examples in first-order logic

4.1 Introduction

An excellent introduction to the subject of machine learning can be found in [Michalski et al. 86]. According to Michalski, learned knowledge is inherently conjectural, i.e. any knowledge created by generalization from specific observations or by analogy to known facts cannot in principle be proven correct, although it may be disproved. He claims that all scientific human information processing activities are oriented toward determining adequate and simple descriptions or explanations of surrounding environment and phenomena [Michalski 74]. The ability to create the simplest descriptions, using only the “most significant” concepts, and disregard the “irrelevant details”, is highly regarded and considered evidence of intelligence.

One of the important potential applications of inductive learning programs, according to Michalski [Michalski 77], is in knowledge-based systems, which have a potential for wide application. Due to the error-prone and tedious nature of programming knowledge by hand into the computer, there is a need for developing new efficient ways of introducing knowledge into machines. Inductive programs could determine production rules from specific examples of decisions or transformations, optimize a given body of rules (by joining a few specific rules into one more general rule or by detecting unnecessary conditions), create “rule models” which compactly describe a given body of rules (and can be useful for identifying missing information or errors in new rules), automatically correct rules in view of new contradictive information, etc. He also believes that inductive programs could be used to help specialists working in applied sciences, e.g. biology, plant pathology, physiology, medicine, etc., in formulating hypotheses explaining data, detecting data patterns in complex numerical data, selecting the most relevant variables describing data, etc.

At present, to make a computer perform a task, one has to write a complete and correct algorithm for that task, and program the algorithm into the computer. These activities involve a tedious and time-consuming effort by specially trained personnel. Current computer systems cannot improve significantly on the basis of

past mistakes, nor can they acquire new abilities by observing and imitating experts. Machine learning research strives to open the possibility of instructing computers in such new ways.

For a comprehensive survey of inductive inference theory, see [Angluin and Smith 83]. They distinguish between inductive inference and learning as follows : to learn is to “gain knowledge, understanding, or skill by study, instruction or experience”. In contrast, induction is defined as “the act, process, or result of an instance of reasoning from a part to a whole, from particulars to general, or from the individual to the universal”.

In what follows, we describe some of the past work in different areas of machine learning. A number of different definitions of learning are outlined, to give the reader a feel for some of the different approaches to learning taken by researchers in the field.

Valiant [Valiant 84] views learning as the process of deducing a program for performing a task, from information that does not provide an explicit description of such a program. The description language used is propositional calculus. A concept is said to be “learned” if a program for recognizing it has been deduced (by means other than the acquisition from the outside of the explicit program). The deduction procedure will output an expression that approximates the expression to be learned with high likelihood. He shows that it is possible to design learning machines that have all of the following three properties :

1) The machines can provably learn whole classes of concepts and these classes can be characterized.

2) The classes of concepts are appropriate and non-trivial for general-purpose knowledge.

3) The computational process by which the machines deduce the desired programs requires a feasible (i.e. polynomial) number of steps.

The learner is assumed to have access to a supply of positive examples which have a probabilistic distribution determined arbitrarily by nature. Valiant demonstrates that the following three classes of Boolean expressions are learnable in polynomial time : (1) conjunctive normal form expressions with a bounded number of literals in each clause; (2) monotone disjunctive normal form expressions, and (3) arbitrary expressions in which each variable occurs just once. The main technical discovery in this paper is that with this probabilistic notion of learning, highly convergent learning is possible for whole classes of Boolean functions. This distinguishes this approach from more traditional ones where learning is seen as a process of “inducing” some general rule from information that is insufficient for a reliable deduction to be made. However, the conjectured existence of some good cryp-

tographic functions that are easy to compute implies that some easy-to-compute functions are not learnable.

In his landmark paper, Gold [Gold 67] defines a language learnability model as the following triple :

1. A definition of learnability
2. A method of information presentation
3. A naming relation which assigns names to languages.

Only one definition of learnability, called *identifiability in the limit*, is considered in Gold's paper. A class of languages is said to be identifiable in the limit with respect to a given language learnability model if there is an algorithm which makes a guess at each time instant about the identity of the language being learned, and after some finite time the guesses are all the same and are a name of the language being learned. Note that the learner does not necessarily know when its guess is correct, and must go on processing information forever because there is always the possibility that information will appear which will force it to change its guess. Two basic methods of information presentation are considered by Gold, namely "text" and "informant". A *text* for a language L is a sequence of strings from L such that every string of L occurs at least once in the text. On the other hand, an *informant* for L can tell the learner whether any string is an element of L, and does so for every possible string over the given alphabet. Two naming relations are considered, namely "tester" and "generator". In both cases the name of a language, i.e. a grammar, is a Turing machine : a *tester* for L is a Turing machine which is a decision procedure for L, and a *generator* for L is a Turing machine which generates L. Note that a tester for L exists if and only if L is recursive, and a generator for L exists if and only if L is recursively enumerable. Gold classifies classes of languages on the basis of whether they are learnable in particular learning models. He shows that the class of context-sensitive languages is learnable from an informant, but that not even the class of regular languages is learnable from a text.

A number of researchers have investigated the problem of synthesizing logic programs from examples. Shapiro [Shapiro 81] describes an incremental inductive inference algorithm for solving model inference problems. The model inference problem is defined as follows : "Given the ability to test observational sentences for their truth in some unknown model M, find a finite set of hypotheses, true in M, that imply all true observational sentences." An example of an inductive inference problem is program synthesis from examples. The task is to infer a program inductively, given examples of its input/output behavior. Shapiro gives two algorithms which derive Horn clause programs given a collection of facts. The first is an enumerative model inference algorithm which enumerates all finite sets of Horn

clauses over a certain language L and tests each of them one by one until a program is found which is neither too strong nor too weak with respect to the known facts. This enumeration approach is very powerful but also extremely inefficient. The second algorithm (incremental model inference algorithm) begins with the conjecture " $T = false$ " and incrementally refines or generalizes T until T is neither too strong nor too weak with respect to the known facts. Shapiro defines a refinement operator for use in the algorithm, and describes a backtracing algorithm for removing refuted hypotheses from T if T is too strong.

In [Sammut and Banerji 86], the authors explore two issues : the role that memory plays in acquiring new concepts, and the extent to which the learner can take an active part in acquiring these concepts. A program MARVIN which learns concepts from examples is described. MARVIN uses a description language very similar to Prolog; therefore for MARVIN, learning a concept is equivalent to synthesizing a logic program. A concept is represented by a set of Horn clauses. Given an example of a concept, MARVIN generates a trial concept by generalizing the example. Then, in order to find out if the trial concept is consistent or not, MARVIN shows the "trainer" (i.e. the user) instances of the concept. If the program can construct an object that does not belong to the target concept but does belong to the trial concept, then the trial concept is inconsistent and a new one must be found. MARVIN has no ability to invent existentially quantified variables other than those derived from the example, nor can it deal with universal quantification. The results obtained by MARVIN are similar to those obtained by Shapiro [Shapiro 81] and Tinkham [Tinkham 90] in that MARVIN can be regarded as a synthesizer of Prolog programs.

Vere [Vere 75] developed a method for inducing concepts which can be described by a conjunction of literals in the predicate calculus, with terms limited to constants and universally quantified variables. The method relies upon a graph representation of a conjunction of literals. An n -ary predicate is represented by a list of $n + 1$ terms, the first of which represents the predicate but is otherwise undistinguished from the remaining n terms of the list, which are the arguments of the predicate. A generalization of a conjunction of literals is obtained by (a) dropping zero or more literals from the conjunction, and (b) replacing constants by variables. A maximal unifying generalization of two conjunctions of literals α and β is defined to be a conjunction of literals γ such that γ is more general than α and more general than β , and such that there is no conjunction of literals γ' such that γ' is more general than both α and β and such that γ is more general than γ' . Vere's method works in a bottom-up fashion in which the input examples are processed one at a time to build the set of conjunctive generalizations. The algorithm for generalizing a pair

of events consists of the following four steps :

1) The literals in each of the two events are matched in all possible ways to generate a set of matching pairs P . By definition, two literals match if they contain the same number of constants and they share at least one common term in the same position.

2) All possible subsets of P are selected such that no single literal of one event is paired with more than one literal in another event. Each of these subsets eventually forms a new generalization of the events.

3) Each subset of matching pairs from Step 2 is extended by adding to the subset additional pairs of literals that did not previously match. A new pair is added to a subset S of P if each literal in that pair is related to some other pair in S by a common constant in a common position.

4) Finally, the resulting set of pairs is converted into a new conjunction of literals by merging each pair to form a single literal.

One problem with this method is the creation of "vacuous literals", such as $(X Y Z)$, i.e. a literal where the predicate as well as all arguments are variables. Such a literal is obviously meaningless but can be generated by the algorithm, which makes it inefficient. Steps 2 and 3 compute a very large number of subsets and it is not clear if the method would be viable in practice. This method will be discussed in more detail in Section 4.7.

The above work is extended in [Vere 78] to learning operators from situation sequences and before-and-after pairs. A relational production has the form $\alpha \rightarrow \beta$, where α and β are conjunctions of literals, called the antecedent and the consequent respectively. The intersection of the sets of literals contained in the antecedent and the consequent is called the context of the production. For the purposes of generalization, a production is regarded as an ordered list of three conjunctions of literals : the context γ , the antecedent α , and the consequent β . Given two productions, their maximal common generalizations are computed by computing the maximal common generalizations of each of their three components using the method presented in [Vere 75] for computing the maximal common generalization of two conjunctions of literals. The process is generalized to a method for computing a minimal set of maximal common generalizations of an arbitrary number of productions. Vere presents four example problems run on the Thoth-p computer implementation of this production generalization theory.

Mitchell [Mitchell 77] describes a method for learning rules from a set of positive and negative training instances with reference to the Meta-DENDRAL program [Buchanan and Mitchell 77]. They use a candidate elimination approach which maintains and modifies a representation of the space of all plausible rule versions.

Their algorithm is guaranteed to find all rule versions consistent with the observed training data, without any backtracking and independent of the order of presentation of the training instances. They define a partial order representing “generality” (or specificity) on rules; a rule R1 is said to be less general (or more specific) than a rule R2 if and only if it is applicable to a proper subset of the instances in which rule R2 will apply. Version spaces are then represented by sets G and S of maximally general versions and maximally specific versions respectively. Each time a new training instance is introduced, all the rules conflicting with that instance are eliminated from the version space. If the new training instance is a negative example, then each element of G which matches the instance must be replaced by a minimally more specific version which does not match the instance. This is done by adding constraints taken from elements in the maximally specific version, and thus remaining consistent with the previous positive examples. The dual operation is performed for positive examples. The system described is not reliable in “noisy” domains as in such a case all rules will be eliminated from the version space.

In [Dietterich and Michalski 83], the authors describe four criteria for evaluating learning methods and apply them to five existing systems for learning from examples. The study focuses on the problem of learning structural descriptions from a set of positive training instances, or methods for finding the maximally-specific conjunctive generalization (MSC-generalization) that characterizes a given class of entities. A conjunctive generalization is defined as a description of a class of objects obtained by forming the conjunction of a group of primitive statements. A MSC-generalization is the most detailed (most specific) description that is true of all the known objects in the class. Since specific descriptions list many facts about the class, the MSC-generalization is the longest conjunctive generalization that still describes all of the training instances. The partially ordered space of descriptions of different levels of generality can be described by indicating what transformations are being applied to change less general descriptions into more general ones. A generalization rule is one which, when applied to a description S_1 , produces a more general description S_2 , i.e. $S_1 \rightarrow S_2$ holds. The generalization rules considered are :

- 1) Dropping condition rule
- 2) Turning constants to variables
- 3) Adding internal disjunctions
- 4) Closing interval
- 5) Climbing generalization tree
- 6) Finding extrema of partial orders (constructive induction rule).

Most existing systems have not implemented constructive induction rules in any general way. Instead, specific procedures are written to generate the new de-

scriptors. Induction methods can be divided into bottom-up (data-driven), top-down (model-driven) and mixed methods. Bottom-up methods process the input events one at a time, gradually generalizing the current set of descriptions until a final conjunctive generalization is computed. Top-down methods search for a small number of conjunctions that together cover all of the input events. First some initial hypotheses are chosen from the partially ordered set of all possible descriptions. If these hypotheses satisfy certain criteria, the search halts. Otherwise the current hypotheses are modified by slightly generalizing or specializing them. Top-down methods have better noise-immunity than bottom-up methods and can easily be extended to cover disjunctions; however, the working hypotheses must be repeatedly checked to determine whether they subsume all of the input events. The selected methods of induction are evaluated in terms of the following four criteria : adequacy of the representation language, rules of generalization implemented, computational efficiency, and flexibility and extensibility.

The problem of computing maximally-specific generalizations of relational descriptions is also examined in [Watanabe and Rendell 90]. The authors describe a search program, called X-search, for finding MSC-generalizations of given structural descriptions. A one-sided approach to Mitchell's version space method [Mitchell 77] is adopted here; the authors compute only the set S of maximally-specific expressions consistent with the given examples, and not the set G of all maximally general expressions consistent with the given examples. Since k -ary predicates can be represented using a combination of unary and binary predicates, the representation used only allows unary and binary predicates. Only existential quantifiers are permitted in their representation. The computation of generalizations is a search in a tree where nodes correspond to relational descriptions.

In a recent paper, Vanlehn [Vanlehn 89] describes an algorithm for specializing overly general concepts. It may happen that in the process of inducing a concept from examples, it is necessary to make the concept less general because the concept matches a negative example. The representation he uses for concepts is a conjunction of positive literals, where all variables are represented existentially. Examples are represented by conjunctions of ground literals. Also, all variables are assumed to designate distinct objects. With this restriction, the set of all generalizations of a concept s corresponds to the set of all subsets of the literals of s . The specialization algorithm described makes use of a bit-vector representation that allows fast parallel bit-vector computation to be used and converts the problem into the set covering problem, which is known to be NP-complete. Despite this, Vanlehn reports that the algorithm seems to work reasonably well in practice.

In [Kodratoff and Ganascia 86], the authors describe algorithms for generalizing

a set of examples. Their generalization algorithm discovers significant links in the examples and expresses them as variable bindings. The first step of generalization is to find for each member E of the learning set a formula E' such that E' can be rewritten as E using transformation rules such that all the E' 's "structurally match" each other. Two formulas are said to "structurally match" if they are identical except for the constants and the variables that instantiate their predicates. The generalization rules used are : climbing the generalization tree and using rewrite rules. First some constant is chosen in each example and replaced by a common variable. Then the occurrence of these newly introduced variables is detected and the fact that these occurrences belong to all the examples is checked. If this is not the case, then an attempt is made to generate them using the provided generalization rules. In general, more than one generalization of two given examples may be obtained, due to the fact that structural matching does not always give a unique solution. The generalizations obtained may be incomparable (i.e. none of these generalizations is more general than any other). The second step of generalization is called the "generalization phase" and involves detecting the common links between variables in all the structurally matching formulas. The representation language is analogous to using literals and conjunction. There are no disjunctions or quantifiers in the examples. Thus the scope of this system is somewhat limited.

In [Kodratoff et al. 85], the same idea is pursued. The authors follow the principle that variable links present in both examples and counterexamples are not very significant, whereas links producing a matching failure are very significant. They present a clustering algorithm based on a "syntactic distance" defined by considering the size and nature of the changes made to a description in order to put it into structural matching with the others. The parameters which are used to measure the syntactic distance between a pair of examples are : the predicates which have to be dropped when generalizing the two examples (using the dropping rules), the predicates which are common to the two examples, the predicates which are introduced by using theorems when generalizing the two examples, and the predicates introduced by the use of idempotency when generalizing the two examples. A partial ordering is associated with each of these four measures and a hierarchy of concepts built based on the resulting partial order relation. The definition of the partial orders allows for the use of heuristics by the user to give a higher weight to some predicates than to others, etc.. The method seems rather complicated and its utility is not demonstrated in the paper.

Mitchell [Mitchell 83] reviews some of the issues involved in learning from examples, in the context of a particular learning program called LEX. LEX illustrates how a program can learn useful heuristics for solving integral calculus problems. Its

design is based on four distinct modules, which are (i) the problem solver, which uses the available heuristics to attempt to solve a given problem within allocated time and resources; (ii) the critic, which analyzes the search tree generated by the problem solver, to produce a set of positive and negative training instances from which heuristics will be inferred; (iii) the generalizer, which proposes and refines general heuristics by generalizing from the training instances provided by the critic; and (iv) the problem generator, which generates practice problems for the system. Alternative descriptions of a heuristic are called the version space of the heuristic. Although LEX has been able to improve its performance at solving integral calculus problems by a few orders of magnitude when practice problems were provided by hand, results have not been nearly so encouraging when LEX was provided with practice problems by the problem generator.

From the above, it is clear that the approaches taken to learning by different researchers differ widely both in their methodology as well as in the representation language used. The subject of learning has been divided into a number of different fields, such as learning from examples, learning from observation, learning from instruction, and so on. We will study learning from examples. This topic itself can be subdivided into two categories : learning from positive examples and learning from positive and negative examples (we will deal with the former). Negative examples are also known as counterexamples. When learning from positive examples, a concept must be found which describes all of the input examples. On the other hand, when counterexamples are also provided, the concept to be learned must not only describe all the input (positive) examples, but must also contradict all the input counterexamples or negative examples. In other words, we must find a concept which is a logical consequence of all the positive examples (called the completeness condition) and which is not a logical consequence of any of the negative examples (called the consistency condition).

4.2 Motivation

The specific learning problem which we will be addressing in this chapter is concerned with learning characteristic descriptions from examples. A characteristic description is a description of a collection of objects, situations or events which states facts that are true of all objects in the class. More formally, a statement S is a description of objects O_1, O_2, O_3, \dots if

$$\begin{aligned} O_1 &\rightarrow S, \\ O_2 &\rightarrow S, \\ O_3 &\rightarrow S, \end{aligned}$$

and so on, where \rightarrow denotes logical implication.

A critical issue in machine learning is the choice of a representation language, in which descriptions of situations and the situations themselves are represented. One of the impediments to the progress of research in the field of machine learning has been the diversity of representation languages used, making it laborious to compare and combine research efforts. There is a tradeoff to be made between the complexities of the language chosen and the resulting complexity of the learning process. The complexity of learning descriptions expressed in a rich language is tremendously higher than that of learning descriptions expressed in a more restricted language. This is due to the larger space of possible descriptions in a richer description language. For this reason, much of the previous work on learning from examples used specialized representation languages or subsets of first-order logic (such as conjunctions of literals, ground formulas, formulas without existential quantifiers, etc.) or Boolean logic as representation language. So far, no methods have been proposed for learning from concepts expressed in full first-order logic, with quantifiers and functions.

We present a learning methodology which uses first-order logic as representation language. We feel that using first-order logic to represent examples allows our method to have a much wider applicability due to the power of first-order predicate calculus and its widespread use in a number of different fields. We will show that due to the choice of first-order logic as a representation language, this method can be applied in a number of different fields where first-order logic is used. In particular, we will show that this algorithm can be used very effectively in conjunction with our method for deriving loop invariants for program verification. It can also be applied in traditional areas like the blocks world, where many similar algorithms in the field have been applied. We will then show that the performance of this algorithm compares favorably with others in the same field.

Following the terminology used in [Dietterich and Michalski 83], our algorithm performs concept acquisition, i.e. given examples $E1$ and $E2$ in first-order logic notation, the algorithm attempts to discover a concept EX such that $E1 \rightarrow EX$ and $E2 \rightarrow EX$, and such that EX captures all the features common to both the examples; the concepts and examples are expressed in first-order logic. This is a problem of discovering a logical consequence of two formulas in first-order logic, and will be achieved by using the resolution and unskolemization method of Chapter 2. In this chapter, we will explore in more detail how some of the nondeterminism of the

unskolemization algorithm can be done away with by looking for common features between $E1$ and $E2$. These common features are used to decide which functions will be unskolemized, i.e. which functions will be replaced by existentially quantified variables. Thus these common features will guide the marking of the clauses for unskolemization. Recall that this marking was performed nondeterministically in the unskolemization algorithm.

Note that it may happen that more than two examples are available to us. In such a case, the algorithm can be applied first to two of the examples, then to the concept learned from these examples and the next example, and so on.

4.3 Role of bias in learning

The examples which are presented to a learning program are also called *training instances*. Given a collection of training instances, a *bias* is the set of all factors that collectively influence the selection of the concept learned [Utgoff 86]. Learning from concepts can be regarded as a function of two arguments : the training instances and the bias for concept selection. Thus the choice of a bias is crucial, since it guides the learning program to make a selection from the available concepts.

Now, given a number of examples, there can be an infinite number of logical consequences of these examples. If we are also given counterexamples, then there is a natural way of limiting the candidate solutions, since an additional constraint is imposed on these solutions, namely the consistency condition. However, in the presence of only positive examples with no counterexamples, we must find some other way of imposing a limit on how general a description can be. In other words, a suitable bias must be decided upon. There have been a number of solutions suggested to this problem in the past. One solution is to require that the concept generated be the longest conjunctive statement satisfying the completeness condition [Vere 75], [Hayes-Roth and McDermott 78]. Their representations did not allow disjunctive concepts. Another way is to require that the description not exceed a given degree of generality, which can be measured in several ways. One such method, suggested in [Stepp 78], is to use the ratio of all distinct events which could potentially satisfy the description to the number of positive examples. The bias we use for concept selection is to select a concept which is as specific as possible, subject to a number of different constraints. The motivation for these constraints is explained below. The meaning of "specific" here is similar to the meaning stated earlier for MSC-generalizations.

Our method essentially uses a graphical representation of the clauses in each example, and tries to match pairs of clauses (taking one from each example) in such

a way that the two clauses contain at least one predicate in common. Then the arguments of the predicates of these clauses are matched and a “marking” for the two examples chosen (this “marking” designates which functions will be replaced by existential variables) using a maximum weight matching algorithm for bipartite graphs to determine the marking which will give a learned concept containing as much detail as possible. The unskolemization of the marked formulas then proceeds according to the unskolemization algorithm in Chapter 2.

Let us try to motivate the above ideas. Suppose we are given two input examples. These two examples are compared, and we try to find pairs of clauses C_1 and C_2 , one belonging to each example, such that the intersection of the sets of predicates in each clause is non-empty. The reason for this is that we are trying to find common features of the two examples. For example, if there is a red object named “ a ” in one example, and a red object named “ b ” in the other example, then the two examples would contain clauses :

$\{red(a)\}$ and $\{red(b)\}$ respectively.

We would like to discover this common feature and state that there exists a red object in both examples. In this particular case, the two clauses being compared contain the predicate “ red ” and no other. However, it may happen that we have the following clauses, one taken from each example :

$\{red(a) \vee circle(a)\}$ and $\{red(b)\}$.

Then we would still like to be able to discover the following common feature between these two examples :

$\exists x(red(x) \vee circle(x))$.

This explains why we choose to impose the condition that the intersection of the sets of predicates in each clause should be non-empty if these two clauses are taken from different examples and are to be considered for generalization.

The enquiring reader may ask : why impose any restrictions at all? Why not just try to generalize any pair of clauses, one taken from each example? For instance, given the following two clauses, one from each example :

$\{red(a)\}$ and $\{rectangle(b)\}$,

we could generalize these by the statement

$\exists x(red(x) \vee rectangle(x))$.

It can easily be seen that this statement logically follows from both the given clauses. However, continuing this line of thought to its logical conclusion, why not just let the concept to be learned trivially be the logical disjunction of the two input examples? This disjunction is the most specific logical consequence of the two examples. But we want to do more than just take the logical disjunction of the two examples, because taking the logical disjunction of the two examples is trivial

and conveys no new or useful information about the common features of the two examples. We choose to search for the common features between the two examples as explained in the previous paragraphs.

4.4 A method for learning from examples

4.4.1 Definitions, notation and examples

In what follows, we will use terms and concepts from graph theory. A good grasp of the basic tenets of graph theory can be obtained from [Bondy and Murty 76]. All the definitions which will be needed for our purposes will be given in this section.

Definition. A **graph** G is an ordered pair (V, E) consisting of a non-empty set V of vertices and a set E of edges which are unordered pairs of vertices. We say that an edge (a, b) has ends a and b .

Definition. A **bipartite graph** is one whose vertex set can be partitioned into two subsets X and Y , so that each edge has one end in X and one end in Y ; such a partition (X, Y) is called a **bipartition** of the graph.

Definition. Two edges (a, b) and (c, d) of a graph G are said to be **adjacent** in G if and only if either $a = c, a = d, b = c$, or $b = d$.

Definition. Let $G = (V, E)$ be a graph. A subset M of E is called a **matching** in G if its elements are edges and no two are adjacent in G ; the two ends of an edge in M are said to be **matched** under M .

Definition. A graph G is said to be **weighted** if with each edge e of G there is associated a real number $w(e)$, called its **weight**.

Definition. The **weight** of a matching M of a weighted graph is the sum of all the weights of the edges in M .

Definition. A **maximum weight matching** of a graph G is a matching M of G such that there is no matching M' of G such that the weight of M' is greater than that of M .

Definition. Let C be a clause. Then $pred(C)$ is the set of all the predicate symbols, along with their signs and arities, which occur in C .

For example, if $C = \{\neg P(x), Q(y, a)\}$, then $pred(C) = \{\neg P^1, Q^2\}$; here the superscripts on the predicate symbols are used to indicate their arities.

Before plunging into the details of the algorithm, we give below a few examples to illustrate the issues involved in finding common features between examples and to motivate the algorithm.

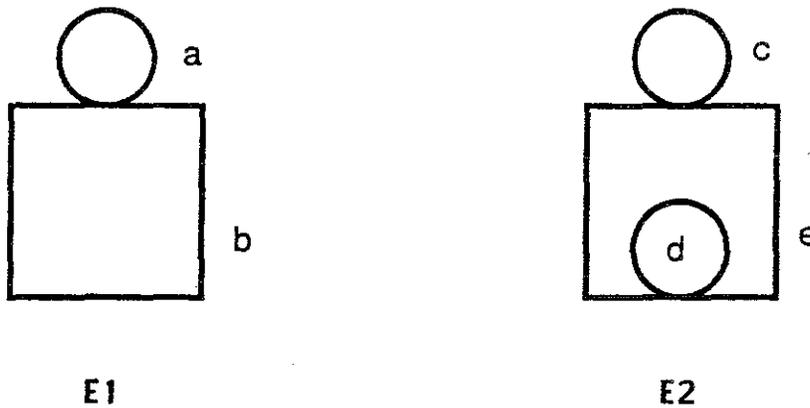


Figure 4.1 Blocks for Example 4.1

Example 4.1 This example is a very simple one in which we are given two examples from the blocks world illustrated in Figure 4.1 (this example is taken from [Dietterich and Michalski 83], Figure 3-1, p. 51). We can represent the two examples $E1$ and $E2$ in first-order logic notation as follows :

$$E1 = circle(a) \wedge square(b) \wedge ontop(a, b),$$

$$E2 = circle(c) \wedge circle(d) \wedge square(e) \wedge ontop(c, e) \wedge inside(d, e),$$

where the constants a through e are used to represent the various objects in the two examples. By comparing $E1$ and $E2$, we can immediately see that a common feature of the two examples is that both have a circle on top of a square. $E1$ contains the following literals :

$$circle(a) \wedge square(b) \wedge ontop(a, b),$$

and $E2$ contains the corresponding literals :

$$circle(c) \wedge square(e) \wedge ontop(c, e).$$

Therefore by matching a with c and b with e , we can get the following consequence of $E1$ and $E2$:

$$\exists X \exists Y (circle(X) \wedge square(Y) \wedge ontop(X, Y)),$$

which is a concept learned from $E1$ and $E2$. The informal meaning of “matching” two arguments is that we unskolemize them and replace them by the same existential variable. Thus here, a and c were “matched” together, and we replaced both of them by the same existential variable X ; similarly b and e were “matched” together

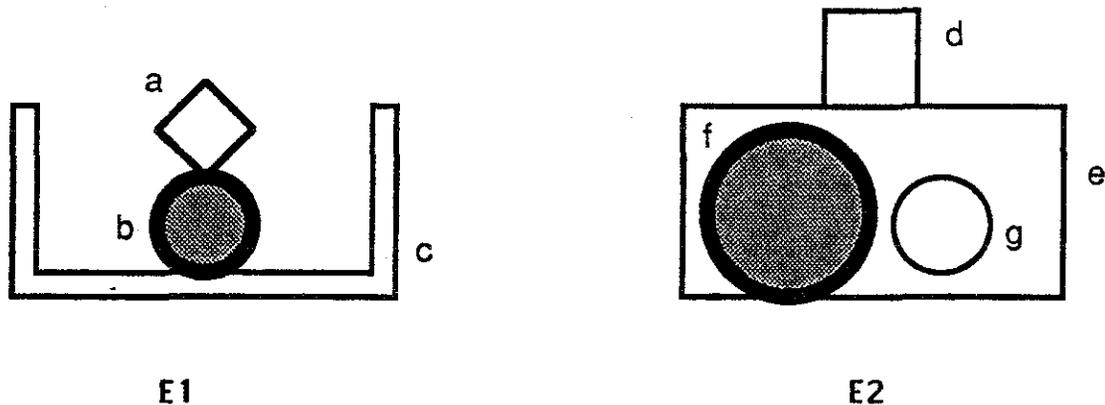


Figure 4.2 Blocks for Example 4.2

and replaced by the same existential variable Y . The concept was obtained by finding some predicates which $E1$ and $E2$ have in common, and unskolemizing the conjunction of those predicates. •

Example 4.2 This example is more detailed than the previous one. We have the following examples $E1$ and $E2$, again from the blocks world, shown in Figure 4.2 :

$E1 = \text{diamond}(a) \wedge \text{circle}(b) \wedge \text{box}(c) \wedge \text{blank}(a) \wedge \text{shaded}(b) \wedge \text{blank}(c) \wedge \text{thickrim}(b) \wedge \text{ontop}(a, b) \wedge \text{ontop}(b, c) \wedge \text{small}(a) \wedge \text{small}(b) \wedge \text{large}(c),$

$E2 = \text{circle}(f) \wedge \text{circle}(g) \wedge \text{square}(d) \wedge \text{rectangle}(e) \wedge \text{blank}(d) \wedge \text{blank}(e) \wedge \text{shaded}(f) \wedge \text{blank}(g) \wedge \text{thickrim}(f) \wedge \text{ontop}(d, e) \wedge \text{inside}(f, e) \wedge \text{inside}(g, e) \wedge \text{small}(d) \wedge \text{large}(e) \wedge \text{large}(f) \wedge \text{small}(g).$

We also have the following axioms concerning geometric shapes :

1. $\forall X(\text{diamond}(X) \rightarrow \text{polygon}(X))$
2. $\forall X(\text{square}(X) \rightarrow \text{polygon}(X))$

or, in clause form :

- 1'. $\{\neg \text{diamond}(X), \text{polygon}(X)\}$
- 2'. $\{\neg \text{square}(X), \text{polygon}(X)\}.$

We now need to find similar features in $E1$ and $E2$. We note the following points :

(i) a and d have three features in common : they are both small, blank objects and are both on top of some other object. Also, by resolution from the axioms, we can deduce that a and d are both polygons.

(ii) Both c and e are large, blank objects, on top of which some object has been placed.

(iii) The circle b has features in common with both f and g : all three are circles, b and f are shaded objects with thick rims, and b and g are both small. Thus b has more features in common with f than with g .

(iv) There are a great number of other features which $E1$ and $E2$ have in common; for example, a and g are both blank objects, c and d are also both blank objects, etc..

If we want to generate a concept containing as many common features as possible of the two examples, we can get, by pairing a with d , b with f , and c with e , the following concept :

$$\exists X \exists Y \exists Z (\text{polygon}(X) \wedge \text{blank}(X) \wedge \text{small}(X) \wedge (\text{ontop}(X, Y) \vee \text{ontop}(X, Z)) \wedge \text{circle}(Y) \wedge \text{thickrim}(Y) \wedge \text{shaded}(Y) \wedge \text{large}(Z) \wedge \text{blank}(Z) \wedge (\text{ontop}(X, Z) \vee \text{ontop}(Y, Z))).$$

(Note : the exact procedure by which these results were obtained will be explained later. At the moment we are simply trying to illustrate the method we will be using.) •

Recapitulating, we have exposed the following issues which arise when detecting common features between examples :

(i) It may be possible to detect more common features between examples by performing resolutions between each example and the axioms given than by comparing the examples without performing any resolutions.

(ii) Although there are many ways to match the arguments of literals before unskolemizing, some matches are “better” than others in the sense that more common features are detected with those matches.

We now explain the method which will be used in the learning algorithm in order to find the “best” possible match between arguments of predicates in the two examples. We will use graphical representations of clauses and arguments and from these graphs we shall determine an optimum matching. We illustrate the method by using the same examples as in Example 4.2.

First we build the **clause graph** G_c for these two examples. G_c is a bipartite graph with bipartition (X_c, Y_c) , where X_c is the set of clauses representing the example $E1$, obtained by performing resolutions between $E1$ and the set of axioms, and Y_c is the set of clauses representing the example $E2$, obtained by performing resolutions between $E2$ and the set of axioms. The edges E_c of the clause graph are obtained by introducing an edge (C_1, C_2) into E_c if and only if C_1 is a clause in X_c , C_2 is a clause in Y_c , and $\text{pred}(C_1) \cap \text{pred}(C_2) \neq \emptyset$. The clause graph for E_1 and E_2 is shown in Figure 4.3.

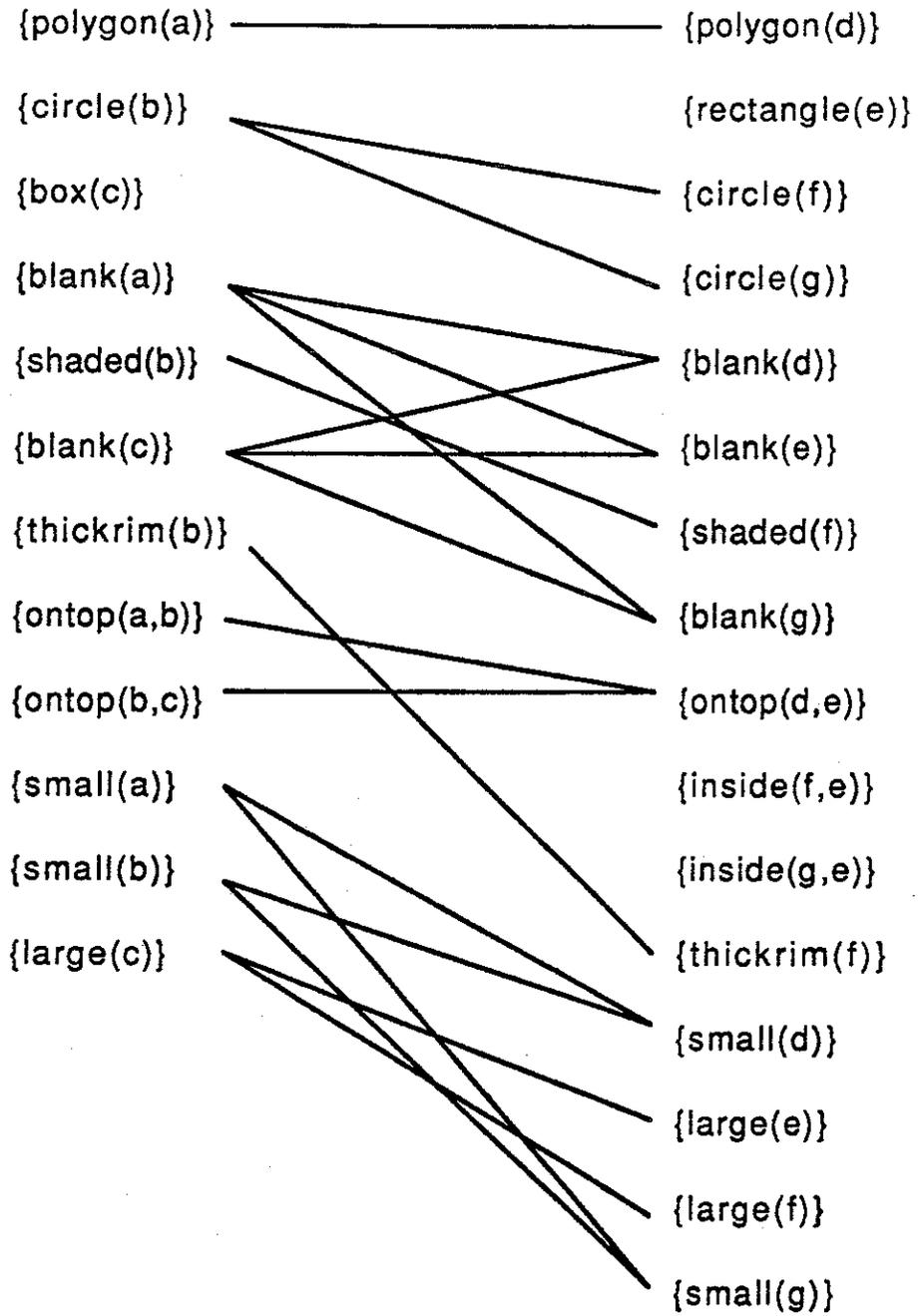


Figure 4.3 Clause graph for Example 4.2

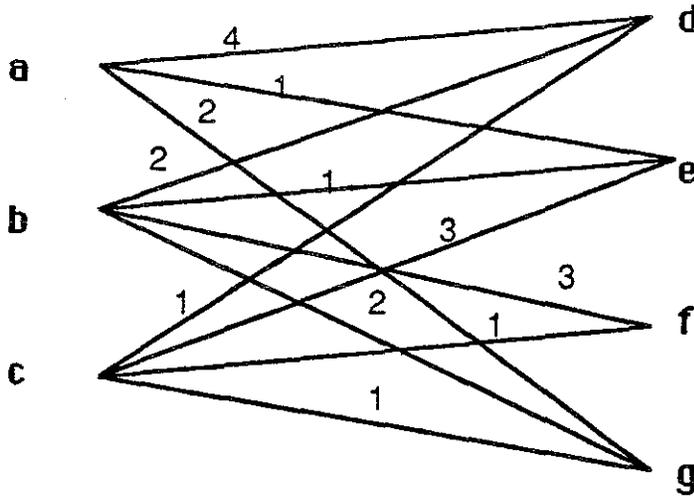


Figure 4.4 Argument graph for Example 4.2

The edges between clauses in this graph indicate potential generalizations; e.g. here the edge between $polygon(a)$ and $polygon(d)$ tells us that we could unskolemize both these clauses to give the formula $\exists X polygon(X)$. Here the constants a and d would be unskolemized and replaced by the same existentially quantified variable X , i.e. a and d are being paired or matched together. We need to find a matching of the arguments in E_1 and E_2 which will give us a learned concept with as many of the common features of E_1 and E_2 as possible. This is done by constructing the argument graph G_a for E_1 and E_2 .

Here, G_a is a weighted bipartite graph, with bipartitions X_a and Y_a . For every edge (C_1, C_2) of E_c , we put the arguments of the literals of C_1 in X_a , the arguments of the literals of C_2 in Y_a , and we add an edge of weight 1 between corresponding arguments of corresponding predicates (i.e. identical predicates) of these clauses. If an edge already exists between the two arguments, then its weight is incremented by 1. The construction of these two graphs becomes more complicated if either of the two examples contain universally quantified variables or if arguments of predicates are functions of arity one or more. This will be explained later. The argument graph for our example is shown in Figure 4.4.

Here there is an edge of weight 4 between a and d , because there are 4 edges in the clause graph between clauses containing literals which have a and d in

corresponding argument positions (these edges are $(\{polygon(a)\}, \{polygon(d)\})$, $(\{blank(a)\}, \{blank(d)\})$, $(\{small(a)\}, \{small(d)\})$, $(\{ontop(a, b)\}, \{ontop(d, e)\})$). The other edges are similarly derived. The edges of the argument graph thus show which arguments can be paired, and the weights on these edges are in some sense proportional to the “goodness” of each pairing. Thus we could pair a with either d, e or g , but the pairing with d would be the best since a and d have four features in common (the weight of edge (a, d) is 4), whereas a and e have one feature in common (both are blank objects) and a and g have two features in common (both are blank and small objects).

Thus we need to find a one-to-one mapping of a subset of X_a with a subset of Y_a such that an argument α is mapped to an argument β only if $(\alpha, \beta) \in E_a$, where E_a is the set of edges of the argument graph. We choose to have a one-to-one mapping rather than a many-to-one or a many-to-many mapping because such generalizations usually do not contribute to the detail of the concept generated, they are sometimes meaningless, and their generation is computationally expensive. Also, this mapping should be such that the sum of the weights of the edges between the mapped elements is a maximum. But this is exactly the problem of finding a maximum weight matching M_a in the weighted bipartite graph G_a . For our examples, this matching can be shown to be the set $\{(a, d), (b, f), (c, e)\}$.

If we use this mapping, we need to unskolemize the clauses of the two examples which contain literals with the same predicate containing matched arguments in common argument positions. We do this as follows. From the clause graph, we form a new subgraph by keeping only those clauses C which have at least one edge e connecting them to another clause D , such that clauses C and D contain at least one predicate in common which has an argument α in the i^{th} position in C , and which has an argument β in the i^{th} position in D (for some positive integer i) such that $(\alpha, \beta) \in M_a$. We also keep all edges which satisfy the conditions satisfied by edge e above. This subgraph is shown in Figure 4.5.

Let us mark the arguments to be unskolemized as follows. Since a and d are matched arguments, we will replace a and d by $X \leftarrow a$ and $X \leftarrow d$ respectively; similarly we replace b and f by $Y \leftarrow b$ and $Y \leftarrow f$ respectively; and finally we replace c and e by $Z \leftarrow c$ and $Z \leftarrow e$ respectively. The marking “ $X \leftarrow a$ ” indicates that the argument “ a ” will eventually be replaced by the existentially quantified variable “ X ”, and a similar meaning holds for the other marked arguments. The resulting marked formulas are :

$$E_1 = polygon(X \leftarrow a) \wedge blank(X \leftarrow a) \wedge small(X \leftarrow a) \wedge ontop(X \leftarrow a, Y \leftarrow b) \wedge circle(Y \leftarrow b) \wedge thickrim(Y \leftarrow b) \wedge shaded(Y \leftarrow b) \wedge blank(Z \leftarrow c) \wedge large(Z \leftarrow c) \wedge ontop(Y \leftarrow b, Z \leftarrow c),$$

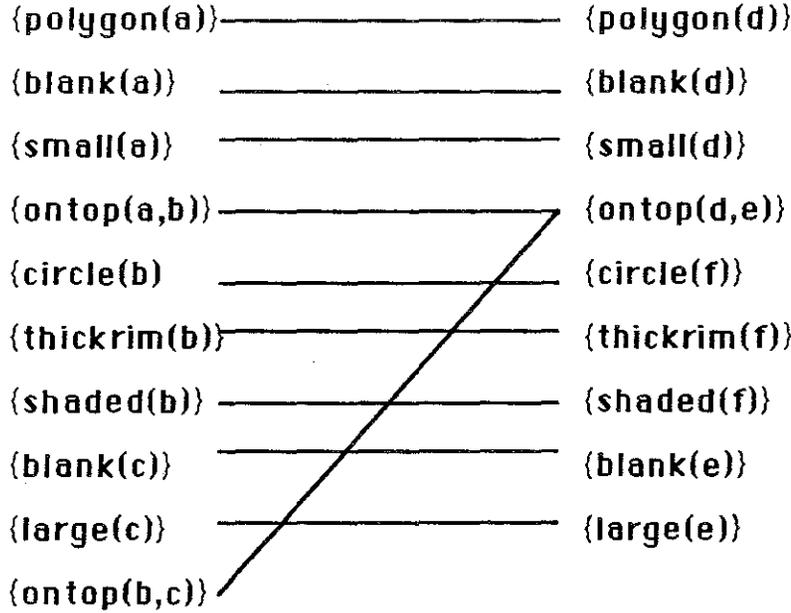


Figure 4.5 Subgraph of clause graph for Example 4.2

$E_2 = \text{polygon}(X \leftarrow d) \wedge \text{blank}(X \leftarrow d) \wedge \text{small}(X \leftarrow d) \wedge \text{ontop}(X \leftarrow d, Z \leftarrow e) \wedge \text{circle}(Y \leftarrow f) \wedge \text{thickrim}(Y \leftarrow f) \wedge \text{shaded}(Y \leftarrow f) \wedge \text{blank}(Z \leftarrow e) \wedge \text{large}(Z \leftarrow e)$.

Note that this method of marking E_1 and E_2 corresponds to the marking which is performed in Step 2 of the unskolemization algorithm of Chapter 2. We now take the conjunction of the pairwise disjunctions of the clauses which have edges connecting them in Figure 4.5. We get the following formula :

$EX = (\text{polygon}(X \leftarrow a) \vee \text{polygon}(X \leftarrow d)) \wedge (\text{blank}(X \leftarrow a) \vee \text{blank}(X \leftarrow d)) \wedge (\text{small}(X \leftarrow a) \vee \text{small}(X \leftarrow d)) \wedge (\text{circle}(Y \leftarrow b) \vee \text{circle}(Y \leftarrow f)) \wedge (\text{thickrim}(Y \leftarrow b) \vee \text{thickrim}(Y \leftarrow f)) \wedge (\text{shaded}(Y \leftarrow b) \vee \text{shaded}(Y \leftarrow f)) \wedge (\text{blank}(Z \leftarrow c) \vee \text{blank}(Z \leftarrow e)) \wedge (\text{large}(Z \leftarrow c) \vee \text{large}(Z \leftarrow e)) \wedge (\text{ontop}(X \leftarrow a, Y \leftarrow b) \vee \text{ontop}(X \leftarrow d, Z \leftarrow e)) \wedge (\text{ontop}(Y \leftarrow b, Z \leftarrow c) \vee \text{ontop}(X \leftarrow d, Z \leftarrow e))$.

Replacing arguments of the form " $X \leftarrow a$ " by the existential variable X , we get $EX = \exists X \exists Y \exists Z (\text{polygon}(X) \wedge \text{blank}(X) \wedge \text{small}(X) \wedge \text{circle}(Y) \wedge \text{thickrim}(Y) \wedge \text{shaded}(Y) \wedge \text{blank}(Z) \wedge \text{large}(Z) \wedge (\text{ontop}(X, Y) \vee \text{ontop}(X, Z)) \wedge (\text{ontop}(Y, Z) \vee \text{ontop}(X, Z))$



Figure 4.6 Clause graph for Example 4.3

$ontop(X, Z))$.

EX is a concept learned from the two given examples.

Note that the unskolemization process conforms to the unskolemization algorithm of Chapter 2, except that the arguments of clauses are marked deterministically, by using the information gathered from the clause and argument graphs.

We now describe a refinement to the above procedure when universal variables occur in at least one of the examples. The refinement will first be illustrated in the following example.

Example 4.3 This is a short example to illustrate the process of matching arguments when the arguments of some literals are functions of arity one or more and when some arguments which are being matched are universally quantified variables. Let E_1 and E_2 be the following two examples :

$$\begin{aligned}
E_1 &= P(f(c), f(f(c), g(e))) \wedge Q(a, h(f(a))) \\
E_2 &= \forall W (P(f(b), f(f(b), d)) \wedge Q(W, h(f(W))))).
\end{aligned}$$

There are no axioms for this example. The clause graph for these two examples is shown in Figure 4.6. To construct the argument graph, we need to look at corresponding arguments in clauses of E_1 and E_2 which are connected by edges in the clause graph. First consider the corresponding arguments of the predicate P . The first argument of P in E_1 is $f(c)$ and the first argument of P in E_2 is $f(b)$. Instead of putting $f(c)$ and $f(b)$ in X_a and Y_a respectively of the argument graph, we will put c in X_a and b in Y_a , and add an edge between c and b . Similarly, the second argument of P in E_1 is $f(f(c), g(e))$ and the second argument of P in E_2 is $f(f(b), d)$. We therefore increment the weight of the edge (c, b) in G_a by 1, and we add an edge between $g(e)$ and d . Continuing like this, we get the argument graph shown in Figure 4.7. Finding a maximum weight matching for this argument graph is

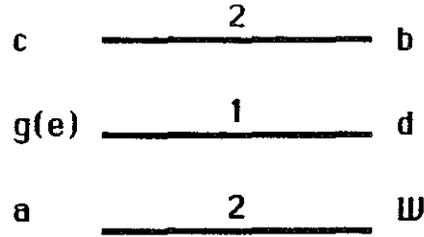


Figure 4.7 Argument graph for Example 4.3

straightforward : the maximum weight matching is $M_a = \{(c, b), (g(e), d), (a, W)\}$. We mark the clauses of E_1 and E_2 for unskolemization as explained in Example 4.2 as follows :

$$E_1 = P(f(X \leftarrow c), f(f(X \leftarrow c), Y \leftarrow g(e))) \wedge Q(Z \leftarrow a, h(f(Z \leftarrow a)))$$

$$E_2 = P(f(X \leftarrow b), f(f(X \leftarrow b), Y \leftarrow d)) \wedge Q(Z \leftarrow W, h(f(Z \leftarrow W))).$$

Taking the conjunction of the pairwise disjunctions of the clauses which have edges connecting them in the clause graph, we get :

$$EX = (P(f(X \leftarrow c), f(f(X \leftarrow c), Y \leftarrow g(e))) \vee P(f(X \leftarrow b), f(f(X \leftarrow b), Y \leftarrow d))) \wedge (Q(Z \leftarrow a, h(f(Z \leftarrow a))) \vee Q(Z \leftarrow W, h(f(Z \leftarrow W)))).$$

Unskolemizing EX by performing Steps 4 through 7 of the unskolemization algorithm, we get :

$$EX = \exists X \exists Y \exists Z (P(f(X), f(f(X), Y)) \wedge Q(Z, h(f(Z)))).$$

However, there is a way to obtain a more specific concept here. Note that a and W were matched together here, and W is a universally quantified variable. W and a were unskolemized and replaced by the existential variable Z . However, W could have been instantiated to a , and the edge (a, W) of weight 2 would have been replaced by the edge (a, a) of weight 2. Then during unskolemization, the arguments "a" in each example could have remained unchanged, and using the same procedure as above, we would have obtained the following concept :

$$EX' = \exists X \exists Y (P(f(X), f(f(X), Y)) \wedge Q(a, h(f(a))))$$

which is more specific than EX (note that $EX' \rightarrow EX$) since the arguments of the predicate Q are a and $h(f(a))$ instead of Z and $h(f(Z))$ for an existential variable Z . •

The above example suggests a refinement to the previously described method for constructing argument and clause graphs. Whenever there is an edge in E_a (i.e. in the argument graph) connecting a universally quantified variable X and a function symbol f , the universally quantified variable should be instantiated to that function, and appropriate edges should be added to the argument graph.

Another way of instantiating a universally quantified variable X is to instantiate X to a function b if there is an edge connecting X to some function a , and there is an edge connecting a to b . The utility of this is demonstrated as follows : suppose

$$E1 = \forall X(P(X) \wedge Q(b)),$$

$$E2 = P(a) \wedge Q(a).$$

Here in the argument graph, there is an edge joining X and a , and another edge joining a and b ; therefore we will instantiate X to b . Then the first example will contain the clauses

$$\{P(b)\} \text{ and } \{Q(b)\},$$

and the second example will contain the clauses

$$\{P(a)\} \text{ and } \{Q(a)\},$$

from which we can deduce

$$\exists Y(P(Y) \wedge Q(Y))$$

as part of the learned concept. This would not have been possible without instantiating X to b .

After all these instantiations are performed and appropriate edges between corresponding arguments of corresponding predicates in G_c are added to the argument graph G_a , the clause containing the universally quantified variable X being instantiated in the clause graph should then be added to the clause graph, along with the corresponding edges. That is, if a universally quantified variable argument X occurring in clause C is instantiated to a in the argument graph, then the clause $C\sigma$ will be added to the clause graph, where σ is the substitution $\{X \leftarrow a\}$. Also for every edge e connecting C to a clause D in the clause graph, a new edge e' connecting $C\sigma$ to D will be added to the clause graph.

4.4.2 The learning algorithm

Let E_1 and E_2 be the two sets of clauses representing the two given examples, and let *AXIOMS* be a set of axioms for the given situation. The following algorithm will find a concept EX such that $E_1 \rightarrow EX$ and $E_2 \rightarrow EX$.

Algorithm LEARN(E_1, E_2, AXIOMS)

begin

Choose $X_c \in Res(E_1 \wedge AXIOMS)$;
 Choose $Y_c \in Res(E_2 \wedge AXIOMS)$;
 {COMMENT : The above two steps are performed nondeterministically}
 Rename the variables in all the clauses of X_c and Y_c so that no two clauses
 have any variable in common;
 build_clause_graph(X_c, Y_c, E_c);
 build_argument_graph(X_a, Y_a, E_a);
 augment_graphs_X(X_a, E_a, X_c, E_c);
 augment_graphs_Y(Y_a, E_a, Y_c, E_c);
 maximum_weight_matching(M_a, X_a, Y_a, E_a);
 {COMMENT : This procedure returns a maximum weight matching M_a for
 the argument graph. The algorithm is not given here since suitable algorithms
 can be found in the literature (see for example [Galil 86])}
 $M_c := \{(C_1, C_2) \in E_c \mid \text{the } n^{th} \text{ argument } \alpha_n \text{ of some literal of } C_1 \text{ contains } a \text{ as a subterm and the } n^{th} \text{ argument } \beta_n \text{ of some literal of } C_2 \text{ contains } b \text{ as a subterm in the same position as } a \text{ appears in } \alpha_n \text{ and } (a, b) \in M_a, \text{ for some positive integer } n, \text{ where these two literals have the same predicate } \}$;
 For every edge $(a, b) \in M_a$ do
 if (a and b are distinct) and (a and b are not both variables) then
 replace unmarked occurrences of a and b in M_c by $Z \leftarrow a$ and
 $Z \leftarrow b$ (respectively) (Z is a new variable);
 {COMMENT : Call these occurrences of a and b marked}
 if (a and b are both variables) then
 unify all occurrences of a and b in M_c ;
 $EX := \{C_1 \cup C_2 \mid (C_1, C_2) \in M_c\}$;
 if $EX = \emptyset$ then $EX := true$;
 for every Skolem function α in EX do
 if α is not marked then
 replace all occurrences of α in all literals of EX by $X \leftarrow \alpha$, where
 X is a new variable not occurring elsewhere in any clause;
 Perform Steps 4 through 7 of the unskolemization algorithm for EX ;
end.

The formula EX is the required "learned concept".

procedure build_clause_graph(X_c, Y_c, E_c);

{COMMENT : The clause graph G_c is a bipartite graph with bipartition (X_c, Y_c) ; thus each clause of X_c or Y_c is a vertex of this graph. The set of edges E_c of this graph is formed as follows :}

begin

$E_c := \emptyset$;

for every $C_1 \in X_c, C_2 \in Y_c$ do

if $(pred(C_1) \cap pred(C_2) \neq \emptyset)$ then

$E_c := E_c \cup (C_1, C_2)$

end.

procedure build_argument_graph(X_a, Y_a, E_a);

{COMMENT : Let the bipartition of the argument graph be (X_a, Y_a) , with edge set E_a . Let w be a positive integer weight function for this graph. Initially, X_a, Y_a and E_a are empty sets. The graph G_a is built as follows.}

begin

$X_a := \emptyset$;

$Y_a := \emptyset$;

$E_a := \emptyset$;

For every pair of clauses C_1, C_2 such that $(C_1, C_2) \in E_c$ do

$\{C'_1 := \{L \in C_1 \mid pred(L) \in C_2\}$;

{COMMENT : This set is non-empty since $pred(C_1) \cap pred(C_2) \neq \emptyset$.}

For every literal L in C'_1 do

{Let $K = \{N \mid N \in C_2 \text{ and } N, L \text{ have the same predicate with the same arity and sign}\}$;

for $i := 1$ to *arity of L* do

for every literal N in K do

suppose the i^{th} argument of L is α_i and the i^{th} argument of N is β_i ;

done := false;

repeat

Try to find terms t_1, t_2 which have the same place in α_i and β_i respectively and such that

(($t_1 \neq t_2$) and ((t_1 and t_2 begin with different function letters) or (at least one of them is a variable)))

or (($t_1 = t_2$) and (t_1 and t_2 are functions without any arguments or variables));

if there are no such t_1, t_2 then done := true

```

else
    { $X_a := X_a \cup \{t_1\}$ ;
     $Y_a := Y_a \cup \{t_2\}$ ;
    if  $(t_1, t_2) \notin E_a$  then
        { $E_a := E_a \cup \{(t_1, t_2)\}$ ;
         $w((t_1, t_2)) := 1$ }
    else
         $w((t_1, t_2)) := w((t_1, t_2)) + 1$ ;
    }
until done
}
}
end.

```

procedure `augment_graphs_X`(X_a, E_a, X_c, E_c);

begin

{COMMENT : The following procedure adds instances of certain arguments of clauses to the set X_a of the argument graph, and adds instances of certain clauses to X_c . This is done as demonstrated in and following Example 4.3}

for every variable Z in $X_a \cup Y_a$ do

$I(Z) := \emptyset$;

{COMMENT : $I(Z)$ is a set corresponding to the variable Z which will contain all the functions to which Z will be instantiated during this procedure}

for every variable $Z \in X_a$ do

{for every edge $(Z, a) \in E_a$ where a is a function do

{for every α such that $(Z, \alpha) \in E_a$ do

$E_a := E_a \cup (a, \alpha)$;

$X_a := X_a \cup \{a\}$

$I(Z) := I(Z) \cup \{a\}$;

}

for every pair of edges $(Z, a), (b, a) \in E_a$ where a, b are functions do

{for every α such that $(Z, \alpha) \in E_a$ do

$E_a := E_a \cup \{(b, \alpha)\}$;

$I(Z) := I(Z) \cup \{b\}$;

}

}

{COMMENT : Now augment the clause graph as follows :}

```

for every clause  $C \in X_c$  do
  for every  $\alpha \in I(Z)$  do
     $X_c := X_c \cup C(Z \leftarrow \alpha)$ ;
    {COMMENT : The clause  $C(Z \leftarrow \alpha)$  is just the clause  $C$  with
    every occurrence of  $Z$  replaced by  $\alpha$ .}
  for every edge  $(C, D) \in E_c$  do
    for every  $\alpha \in I(Z)$  do
       $E_c := E_c \cup \{(C(Z \leftarrow \alpha), D)\}$ ;
end.

```

procedure `augment_graphs_Y`(Y_a, E_a, Y_c, E_c);

begin

{COMMENT : The following procedure adds instances of certain arguments of clauses to the set Y_a of the argument graph, and adds instances of certain clauses to Y_c . This is done as demonstrated in and following Example 4.3}

```

for every variable  $Z \in Y_a$  do
  {for every edge  $(a, Z) \in E_a$  where  $a$  is a function do
    for every  $\alpha$  such that  $(\alpha, Z) \in E_a$  do
       $E_a := E_a \cup (\alpha, a)$ ;
       $Y_a := Y_a \cup \{a\}$ 
       $I(Z) := I(Z) \cup \{a\}$ ;
    }
  for every pair of edges  $(a, Z), (a, b) \in E_a$  where  $a, b$  are functions do
    for every  $\alpha$  such that  $(\alpha, Z) \in E_a$  do
       $E_a := E_a \cup \{(\alpha, b)\}$ ;
       $I(Z) := I(Z) \cup \{b\}$ ;
    }
  }

```

{COMMENT : Now augment the clause graph as follows :}

```

for every clause  $C \in Y_c$  do
  for every  $\alpha \in I(Z)$  do
     $Y_c := Y_c \cup C(Z \leftarrow \alpha)$ ;
    {COMMENT : The clause  $C(Z \leftarrow \alpha)$  is just the clause  $C$  with
    every occurrence of  $Z$  replaced by  $\alpha$ .}
  for every edge  $(D, C) \in E_c$  do
    for every  $\alpha \in I(Z)$  do
       $E_c := E_c \cup \{(D, C(Z \leftarrow \alpha))\}$ ;

```

end.

We now illustrate the working of the algorithm with a simple example.

Example 4.4 The following example illustrates the use of the unskolemization algorithm to learn a concept. Let the two given examples be :

$$E1 = \forall x_1 \forall x_2 P(x_1, f(x_1), a, x_2);$$

$$E2 = \forall y_1 \forall y_2 P(y_1, b, g(y_2), y_2).$$

The clause graph consists of just one edge :

$$E_c = \{(\{P(x_1, f(x_1), a, x_2)\}, \{P(y_1, b, g(y_2), y_2)\})\},$$

and the argument graph contains the following four edges, all of weight 1 :

$$E_a = \{(x_1, y_1), (f(x_1), b), (a, g(y_2)), (x_2, y_2)\}.$$

The maximum weight matching for this graph is straightforward to obtain and consists of all the edges in E_a . We now perform the marking step of the learning algorithm and get the following marked set of clauses EX :

$$EX = \{\{P(x_1, w_1 \leftarrow f(x_1), w_2 \leftarrow a, y_2), P(x_1, w_1 \leftarrow b, w_2 \leftarrow g(y_2), y_2)\}\}.$$

Here w_1, w_2 are new variables. We now perform the unskolemization algorithm :

Add universal quantifiers $\forall x_1 \forall y_2$ to the front of EX and get :

$$EX = \forall x_1 \forall y_2 (P(x_1, w_1 \leftarrow f(x_1), w_2 \leftarrow a, y_2) \vee P(x_1, w_1 \leftarrow b, w_2 \leftarrow g(y_2), y_2)).$$

We replace the marked arguments by existentially quantified variables and get :

$$EX = \forall x_1 \forall y_2 P(x_1, w_1, w_2, y_2)$$

where w_1 and w_2 are existentially quantified variables “dependent” on x_1 and y_2 respectively. By “dependent” we mean that the choice of w_1 depends on x_1 , and similarly for w_2 and y_2 . Thus the universal quantifiers for x_1 and y_2 must precede the existential quantifiers for w_1 and w_2 respectively. There are two formulas which satisfy these constraints; these are

$$EX_1 = \forall x_1 \exists w_1 \forall y_2 \exists w_2 P(x_1, w_1, w_2, y_2)$$

$$\text{and } EX_2 = \forall y_2 \exists w_2 \forall x_1 \exists w_1 P(x_1, w_1, w_2, y_2).$$

These two formulas are the concepts learned from the given input examples $E1$ and $E2$. •

4.5 Soundness and complexity

In the previous sections, we gave a detailed description of a learning algorithm and illustrated its use with the help of a number of examples taken from the blocks world and general first-order logic. We will now prove the soundness of the algorithm, i.e. we prove that if two first-order logic formulas E_1 and E_2 are input to the learning algorithm, any formula EX output by the learning algorithm satisfies

$$E_1 \rightarrow EX \text{ and } E_2 \rightarrow EX$$

i.e. EX is a valid concept learned from the two examples.

We will also discuss the complexity of the algorithm.

4.5.1 Soundness

Theorem 4.1 Given two examples E_1 and E_2 as input, the learning algorithm produces a wff EX such that $E_1 \rightarrow EX$ and $E_2 \rightarrow EX$.

Proof : First note that if no two clauses of E_1 and E_2 contain any predicates in common, then EX will just be the value *true*, which satisfies $E_1 \rightarrow EX$, $E_2 \rightarrow EX$.

Initially, we build clause and argument graphs and augment them by taking certain instances of variables. Every clause contained in X_c is true for the example E_1 , and similarly every clause contained in Y_c is true for the example E_2 . The argument graph links corresponding arguments of certain clauses of X_c and Y_c .

In the next step, certain arguments of clauses of E_1 and E_2 are marked for unskolemization by marking functions " f " as " $Z \leftarrow f$ " or by unifying variables. This is done according to the unskolemization algorithm of Chapter 2. Following this, we take the pairwise union of selected clauses from E_1 and E_2 ; we are performing the pairwise disjunction of these clauses. Finally, the resulting formula is unskolemized using the unskolemization algorithm of Chapter 2.

The result is EX , where EX is the unskolemized form of a set of clauses \mathcal{D} (say), where \mathcal{D} is the conjunction of the pairwise disjunction of some clauses (or instances of clauses) from E_1 and E_2 respectively. From results in Chapter 2,

$$E_1 \vee E_2 \rightarrow unsk(\mathcal{D}) = EX;$$

therefore $E_1 \rightarrow EX$, $E_2 \rightarrow EX$. •

4.5.2 Complexity

We now turn to the question of efficiency. It is not really possible to analyze the complexity of performing resolutions, since this is a nondeterministic process. The remainder of the algorithm, which is the portion of the algorithm which builds graphs, performs maximum weight matchings, and unskolemizes the resulting set of clauses, can be performed in polynomial time. A detailed analysis of the time complexity of the algorithm can be found in the appendix. The maximum weight matching step can be performed using the algorithm described in [Galil 86], in time $O(mn \log_{\lceil m/n+1 \rceil} n)$, where m and n are the number of edges and vertices in the argument graph respectively.

4.6 Application to program verification

In the previous sections, we have seen how the learning algorithm can be applied to traditional areas like the blocks world. We now present an extension of the algorithm with a view to applying it to program verification. It will be demonstrated that the process of deriving loop invariants, as described in Chapter 3, can be aided by applying the learning algorithm. As mentioned earlier, the learning algorithm is used to remove the nondeterminism from the unskolemization process of Chapter 2.

Recall that we described the problem of generating loop invariants as one of generating logical consequences of an infinite number of first-order formulas. Very often, it happens that these formulas (or consequences thereof) have structural similarities which can be detected by our learning algorithm. Since our learning algorithm learns a concept implied by two examples expressed in first-order logic, it can be applied to the problem of deriving loop invariants.

Let us point out here that the task of learning loop invariants is simpler than the blocks world problems which we have seen so far in the following respect : when approximations to loop invariants have been derived, there exists a semi-decision procedure which can indicate whether suitable loop invariants have been derived. This semi-decision procedure consists of examining whether the verification conditions for the program are satisfied or not with the derived loop invariants substituted into the verification conditions. Since first-order logic is semi-decidable, so is the above process.

4.6.1 Applying the learning algorithm

The learning algorithm of Section 4.4.2 can be used for deriving loop invariants in the following manner. Recall that in Chapter 3, we characterized the problem of deriving loop invariants as one of deriving logical consequences of an infinite sequence of formulas A_1, A_2, A_3, \dots . The learning algorithm also generates logical consequences of two given formulas, by first performing resolutions between the axioms of the programming language and the two given formulas, and then by detecting common features between them. In other words, the procedure used will be :

Given : formulas A_i, A_j which hold at the entry to a loop;
a set of axioms (called *AXIOMS*) which characterizes the operations of the flowchart programming language;
repeat

LEARN($A_i, A_j, AXIOMS$)

until a suitable loop invariant has been derived.

The only problem with this procedure is that there is no general decision procedure which can tell us if a suitable loop invariant has been derived. This is because the only way to know if a suitable loop invariant has been derived is to prove that all the verification conditions for the program in question are valid for this value of the loop invariant. Validity being only semi-decidable for first-order formulas, we are only sure of getting a definite answer to this question if the loop invariant derived is suitable.

This problem can be tackled by allowing a limited amount of time for proving that all the verification conditions are valid, and assuming that they are not if they cannot be proved to be valid within that time. Although this is not a foolproof solution, it works satisfactorily for a large number of cases. Also, some provers can show that a given set of clauses is not unsatisfiable by providing a model for the set of clauses, that is, a truth assignment which makes all the clauses in the set true (see for example [Lee 90]).

4.6.2 Derivation of loop invariants

The following examples show how the learning algorithm can be used to derive loop invariants.

Example 4.5 The following example arises as a problem of deriving a loop invariant for the loop of the flowchart program shown in Figure 4.8. A loop invariant was derived for this program in Example 3.2 in the previous chapter. We now show how the same problem is tackled using the learning algorithm. The program loop contains variables x and y which change in value every time the loop is traversed. Let A_i be the condition which holds for these variables before the loop is entered for the i^{th} time. Then we have

$$A_1 = (x = 0 \wedge y = b)$$

$$A_2 = (x = a \wedge y = b - 1)$$

$$A_3 = (x = 2 * a \wedge y = b - 2)$$

$$A_4 = (x = 3 * a \wedge y = b - 3)$$

and so on. Let us take the last two formulas and write them in predicate form. This example is also intended to illustrate the importance of choosing the predicate representation for examples carefully. We choose the following representation :

$$A_3 = x_equals(mult(2, a)) \wedge y_equals(minus(b, 2))$$

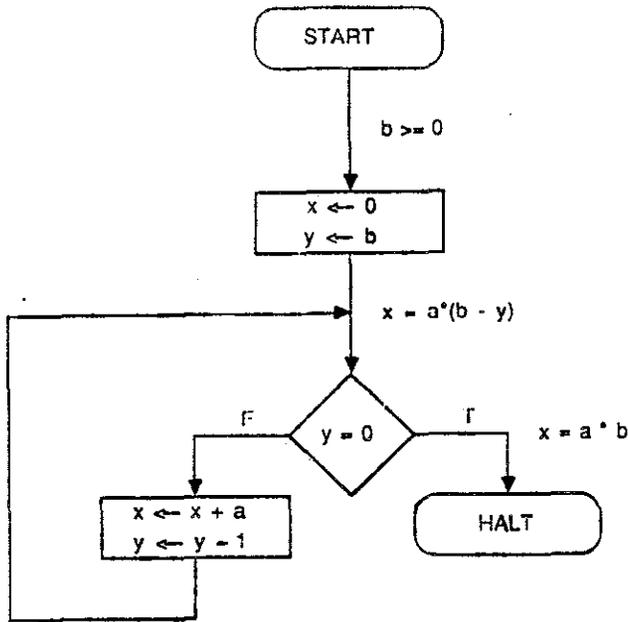


Figure 4.8 Flowchart program for Example 4.5

$$A_4 = x_equals(mult(3, a)) \wedge y_equals(minus(b, 3)).$$

Then the clause graph contains the following two edges :

$$E_c = \{(\{x_equals(mult(2, a)), \{x_equals(mult(3, a))\}), \\ \{y_equals(minus(b, 2)), \{y_equals(minus(b, 3))\}\})\}$$

and the argument graph has the following three edges, with their weights indicated after each edge in brackets :

$$E_a = \{(a, a)(1), (b, b)(1), (2, 3)(2)\}.$$

The maximum weight matching for this graph is straightforward to obtain and consists of all the edges in E_a . We now perform the marking step of the learning algorithm and get the following marked set of clauses EX :

$$EX = \{(\{x_equals(mult(w \leftarrow 2, a)), x_equals(mult(w \leftarrow 3, a))\}, \\ \{y_equals(minus(b, w \leftarrow 2)), \{y_equals(minus(b, w \leftarrow 3))\}\})\}.$$

Here w is a new variable. We now perform the unskolemization. There are no universal quantifiers to be added; the marked arguments are replaced by an existentially quantified variable and we get :

$$EX = \exists w(x_equals(mult(w, a)) \wedge y_equals(minus(b, w))) \quad (I)$$

which is the concept learned here. In arithmetic notation,

$$EX = \exists w(x = w * a \wedge y = b - w).$$

Note that EX is a valid loop invariant for this program loop.

Now consider what results we would have obtained if we had chosen a different predicate representation for the above two examples. Suppose we had written :

$$A_3 = equal(x, mult(2, a)) \wedge equal(y, minus(b, 2)),$$

$$A_4 = equal(x, mult(3, a)) \wedge equal(y, minus(b, 3)).$$

We would have got the following four edges in the clause graph :

$$E_c = \{(\{equal(x, mult(2, a))\}, \{equal(x, mult(3, a))\}), \\ (\{equal(y, minus(b, 2))\}, \{equal(y, minus(b, 3))\}), \\ (\{equal(x, mult(2, a))\}, \{equal(y, minus(b, 3))\}), \\ (\{equal(y, minus(b, 2))\}, \{equal(x, mult(3, a))\})\}$$

and the following seven edges in the argument graph, with their weights indicated after each edge in brackets :

$$E_a = \{(x, x)(1), (y, y)(1), (x, y)(1), (y, x)(1), (a, a)(1), (b, b)(1), (2, 3)(2)\}.$$

We now have two choices for a maximum weight matching for this graph. We will obviously choose the same three edges as before, viz. $\{(a, a), (b, b), (2, 3)\}$. Also we can either choose the two edges $(x, x), (y, y)$ or the two edges $(x, y), (y, x)$. If we choose (x, x) and (y, y) , then we will obtain the same result as we did with the previous representation; however, if we choose the edges (x, y) and (y, x) , then we will get the following concept :

$$EX = \exists w_1 \exists w_2 \exists w_3 (equal(w_1, mult(w_3, a)) \wedge equal(w_2, minus(b, w_3)))$$

which is not of much use to us! This illustrates the importance of choosing a suitable representation for a set of clauses. •

Example 4.6 The program in Figure 4.9 computes the maximum value stored in an array C of n numbers. Suppose that we are trying to compute a loop invariant for the cutpoint B before the entry to the loop. Let A_j be the condition which holds before the loop is entered for the j^{th} time. Then we have

$$A_1 = (max = C(1) \wedge i = 1),$$

$$A_2 = (max = C(1) \wedge max \geq C(2) \wedge i = 2) \vee \\ (max = C(2) \wedge max > C(1) \wedge i = 2),$$

$$A_3 = (max = C(1) \wedge max \geq C(2) \wedge max \geq C(3) \wedge i = 3) \vee \\ (max = C(2) \wedge max > C(1) \wedge max \geq C(3) \wedge i = 3) \vee \\ (max = C(3) \wedge max > C(1) \wedge max > C(2) \wedge i = 3),$$

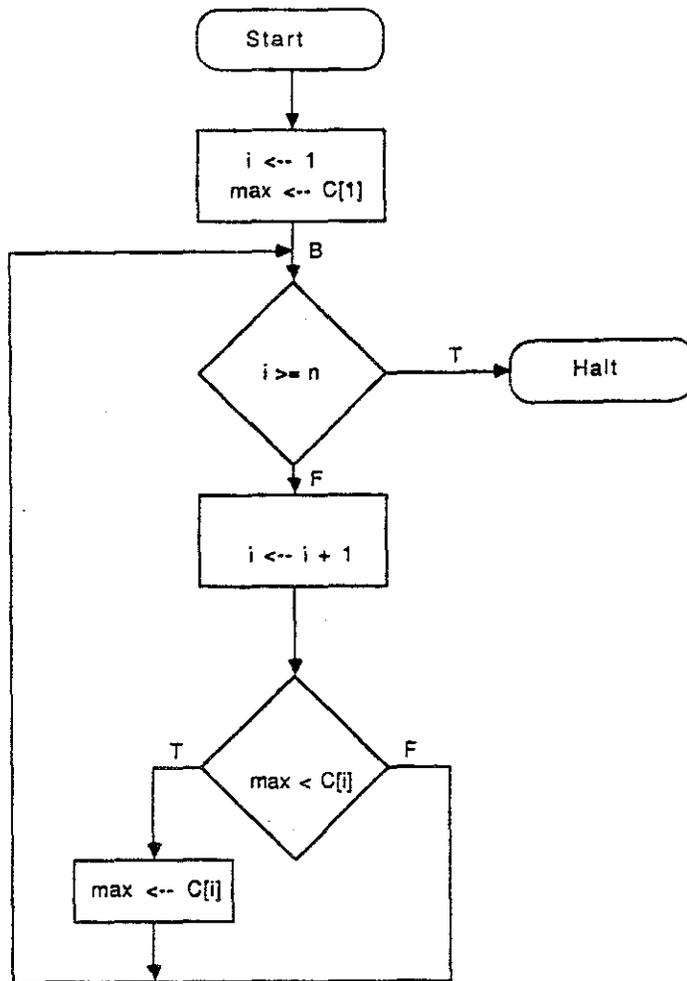


Figure 4.9 Flowchart program for Example 4.6

and so on. Unlike the previous example, we need to further process these formulas before trying to generate a learned concept from them. This is done by performing resolutions between the axioms for the flowchart programming language and the conditions A_2 and A_3 . We choose examples A_2 and A_3 here because the learning algorithm can handle only two input formulas at a time. We could equally well have chosen A_1 and any one of A_2 or A_3 . After a number of resolutions, we get

From A_2 : $\forall k(max \geq C(k) \vee k < 1 \vee k > 2) \wedge (i = 2)$;

From A_3 : $\forall j(max \geq C(j) \vee j < 1 \vee j > 3) \wedge (i = 3)$;

(the resolutions performed can be found at the end of this example). We express these two formulas in predicate form as follows :

From $A_2 : \forall k(max_ge(C(k)) \vee lt(k, 1) \vee gt(k, 2)) \wedge equals_i(2)$;

From $A_3 : \forall j(max_ge(C(j)) \vee lt(j, 1) \vee gt(j, 3)) \wedge equals_i(3)$.

The clause graph for these contains the following two edges :

$$E_c = \{(\{max_ge(C(k)), lt(k, 1), gt(k, 2)\}, \{max_ge(C(j)), lt(j, 1), gt(j, 3)\}), \\ (\{equals_i(2)\}, \{equals_i(3)\})\}$$

and the argument graph has the following three edges, with their weights indicated after each edge in brackets :

$$E_a = \{(k, j)(3), (1, 1)(1), (2, 3)(2)\}.$$

The maximum weight matching for this graph again consists of all the edges in E_a . We perform the matching step of the learning algorithm and obtain :

$$EX = \{(\{max_ge(C(X)), lt(X, 1), gt(X, Y \leftarrow 2), gt(X, Y \leftarrow 3)\}, \\ \{equals_i(Y \leftarrow 2), equals_i(Y \leftarrow 3)\})\}$$

(Here X is a variable obtained by unifying the variables k and j , and Y is a new variable.) We perform the unskolemization algorithm as before, and get

$$EX = \exists Y \forall X ((max_ge(C(X)) \vee lt(X, 1) \vee gt(X, Y)) \wedge equals_i(Y));$$

or, in arithmetic notation,

$$EX = \exists Y \forall X ((max \geq C(X) \vee X < 1 \vee X > Y) \wedge i = Y).$$

EX is a valid loop invariant for this program loop, at cutpoint B .

Resolution proofs for this example :

Derivation of $\forall k((max_ge(C(k)) \vee lt(k, 1) \vee gt(k, 2)) \wedge (i = 3))$ from $A_3 \wedge AXIOMS$:

1. $max = C(1), max > C(1)$	given
2. $\neg(max > X), max \geq X$	axiom
3. $max = C(1), max \geq C(1)$	resolve 1,2
4. $\neg(max = X), max \geq X$	axiom
5. $max \geq C(1)$	resolve 3,4
6. $X = 1, X > 1, X < 1$	axiom
7. $max \geq (C(X)), X > 1, X < 1$	paramodulate 5,6
8. $max = C(2), max > C(2), max \geq C(2)$	given
9. $max = C(2), max \geq C(2)$	resolve 2,8
10. $max \geq C(2)$	resolve 4,9
11. $Y = 2, Y > 2, Y < 2$	axiom
12. $max \geq (C(Y)), Y > 2, Y < 2$	paramodulate 10,11

13. $max = C(3), max \geq C(3)$	given
14. $max \geq C(3)$	resolve 4,13
15. $Z = 3, Z > 3, Z < 3$	axiom
16. $max \geq (C(Z)), Z > 3, Z < 3$	paramodulate 14,15
17. $\neg(W > 1), \neg(W < 2)$	axiom
18. $max \geq (C(X)), \neg(X < 2), X < 1$	resolve 7,17
19. $max \geq (C(Y)), Y > 2, Y < 1$	resolve 12,18
20. $\neg(W > 2), \neg(W < 3)$	axiom
21. $max \geq (C(Y)), Y < 1, \neg(Y < 3)$	resolve 19,20
22. $max \geq (C(Z)), Z < 1, Z > 3$	resolve 16,21
23. $i = 3$	given

The last two clauses constitute the statement : $\forall Z((max \geq (C(Z)) \vee Z < 1 \vee Z > 3) \wedge i = 3)$.

The statement $\forall Z((max \geq (C(Z)) \vee Z < 1 \vee Z > 2) \wedge i = 2)$ can be similarly derived from $A_2 \wedge AXIOMS$. •

4.7 Comparison with other methods

In this section, we compare the performance of our algorithm with that of several similar algorithms in this field. We will point out the differences between our algorithm and those described below, and discuss the significance of these differences, with a view to explaining why our approach is worthy of study.

The algorithms which we will be using for comparison are those of Winston [Winston 75], Vere [Vere 75], Hayes-Roth [Hayes-Roth 78], and Dietterich and Michalski [Dietterich and Michalski 83]. All of these algorithms attack the same problem as described here, namely that of finding characteristic descriptions of situations or objects.

The following discussion compares the five algorithms on the basis of representation language and algorithm used, and performance on a given blocks world problem (taken from [Dietterich and Michalski 83], Figure 3-2, p. 51). A similar comparison was performed by Dietterich and Michalski [Dietterich and Michalski 83] using a number of different algorithms for comparison.

4.7.1 Representation language and learning methodology

In this section we give a brief description of the representation language and the learning methodology used by the authors listed above.

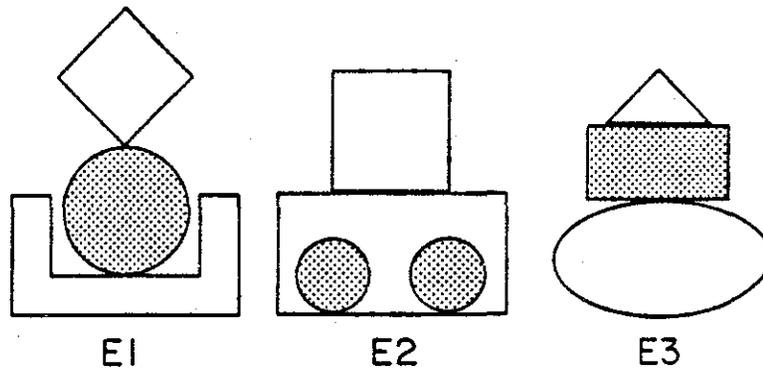


Figure 4.10 Three blocks examples

a) Winston

Winston uses a semantic network to represent examples and background knowledge for the situation at hand. Nodes in the network are used to represent properties of objects, individual examples and links; links in the network represent binary relationships among the nodes. His learning algorithm first obtains a difference description by comparing two input examples and then generalizes this difference description to get one or more new concept descriptions. The difference description is obtained by graph-matching the representations of the two examples, and the generalization is accomplished by creating a network containing the links and nodes of the two examples which matched exactly. The program uses "near-miss" negative examples to find a description of the concept.

Winston's method usually tends to develop most specific generalizations of the input examples, even though he does not precisely define the learning bias used in his algorithm. There is no mechanism for representing disjunctions in his semantic network. He also assumes that examples are chosen by an intelligent teacher who controls the kind of examples used and the order in which they are presented.

b) Hayes-Roth

Hayes-Roth uses parameterized structural representations to represent input events. The parameterized structural representation for the first example E_1 of Figure 4.10 is

{ {medium : a} {diamond : a} {blank : a} {ontop : a, under : b} {medium : b} {circle : b} {shaded : b} {ontop : b, under : c} {large : c} {box : c} {blank : c} }.

An expression such as $\{medium : a\}$ is called a case frame; *medium* is called a case label, and *a* is called a parameter. This representation can easily be translated into first-order logic by writing $\{medium : a\}$ as $medium(a)$, $\{ontop : a, under : b\}$ as $ontop(a, b)$, and so on. This representation can only express conjunctions and does not allow disjunctions.

The learning algorithm works by matching the case frames in the two examples in all possible ways to give a set *A* (two case frames match if all their case labels match). Then a breadth-first search is conducted to find a one-to-one binding for the parameters of one example with the parameters of the other example. Pairs of parameters are then replaced by variables to give a learned concept. The algorithm tries to find most specific generalizations of the two input examples.

c) Vere

Vere's representation consists of a conjunction of "literals" where a literal is a list of constants called "terms" enclosed in parentheses. Using this representation, the first example in Figure 4.10 is represented as follows :

$(medium\ a)(diamond\ a)(blank\ a)(ontop\ a\ b)\ (medium\ b)(circle\ b)\ (shaded\ b)$
 $(ontop\ b\ c)\ (large\ c)\ (box\ c)(blank\ c).$

Every term in a literal is treated uniformly (i.e. "*medium*" is not distinguished from "*a*", etc.). No disjunctions are allowed in this representation.

The learning algorithm works as follows. The literals in the two examples are matched in all possible ways to generate a set of matching pairs. Two literals match if they have at least one common term in the same position and if they contain the same number of terms. After this is done, all possible subsets of this set of matching pairs are chosen so that no literal is paired with more than one literal in the other example. Following this, subsets of matching pairs are augmented by adding pairs *p* of literals such that each literal in *p* is related to some other pair *q* in the same subset by a common term in a common position. Finally, pairs are merged together in each subset to form a single literal by replacing corresponding terms which are not already identical by new variables. The goal of the algorithm is to discover most specific generalizations of the two input examples.

The problem with this representation is that all terms in a literal are treated uniformly, even though the first term of a literal is really a predicate symbol. This gives rise to the following problem : literals consisting only of variables (e.g. $(x\ y)$) can be generated, which are meaningless. The generation of these literals makes the algorithm less efficient. The algorithm allows a many-to-one binding of variables, which is computationally expensive.

d) Dietterich and Michalski

The representation language used here is called VL_2 and is an extension of first-order logic. An example is represented as a conjunction of “selectors”. Each selector contains a function or predicate descriptor (with variables as arguments) and a list of values that the descriptor can take. For example, the first example in Figure 4.10 is represented as :

$$\begin{aligned} & \exists x \exists y \exists z [size(x) = medium] [shape(x) = diamond] [texture(x) = blank] \\ & [ontop(x, y)] [size(y) = medium] [shape(y) = circle] [texture(y) = shaded] \\ & [ontop(y, z)] [size(z) = large] [shape(z) = box] [texture(z) = blank]. \end{aligned}$$

Descriptors are divided into two classes : unary or attribute descriptors, and non-unary or structure-specifying descriptors. The learning algorithm first searches the description space defined by the structure-specifying descriptors, and then searches the attribute descriptor space. The first search is conducted using a form of best-first search in which a set of best candidate descriptions is maintained during the search. The second search is then conducted using a technique similar to that used in the first search. Some heuristic criteria are used to evaluate the value of generalizations; some of these are : maximize the number of input events covered by a generalization, maximize the number of selectors in a generalization, and so on.

The algorithm is specially designed for finding maximally specific generalizations, but it sometimes generates descriptions which are not maximally specific. This is because the search conducted in the structure-only space is allowed to produce less than maximally specific generalizations, because there may exist most specific generalizations in the complete space whose structure-only component is not maximally specific in the structure-only space.

The language used allows internal disjunctions to be represented. An example of an internal disjunction is :

$$[shape(x) = circle \vee rectangle],$$

with the obvious meaning.

4.7.2 Performance comparison

The objective of all the algorithms described so far is to generate a concept which is as specific as possible. We therefore compare the performance of these algorithms on the basis of the concepts learned by them from a set of three examples. The examples, named E_1 , E_2 and E_3 respectively, are shown in Figure 4.10 and can be represented as follows in first-order logic :

$$\begin{aligned} E_1 = & medium(a) \wedge diamond(a) \wedge blank(a) \wedge ontop(a, b) \wedge medium(b) \wedge circle(b) \\ & \wedge shaded(b) \wedge ontop(b, c) \wedge large(c) \wedge box(c) \wedge blank(c), \end{aligned}$$

$E_2 = \text{medium}(d) \wedge \text{square}(d) \wedge \text{blank}(d) \wedge \text{ontop}(d, e) \wedge \text{small}(f) \wedge \text{circle}(f) \wedge \text{shaded}(f) \wedge \text{inside}(f, e) \wedge \text{small}(g) \wedge \text{circle}(g) \wedge \text{shaded}(g) \wedge \text{inside}(g, e) \wedge \text{large}(e) \wedge \text{rectangle}(e) \wedge \text{blank}(e),$

$E_3 = \text{medium}(h) \wedge \text{triangle}(h) \wedge \text{blank}(h) \wedge \text{ontop}(h, j) \wedge \text{medium}(j) \wedge \text{rectangle}(j) \wedge \text{shaded}(j) \wedge \text{ontop}(j, k) \wedge \text{large}(k) \wedge \text{ellipse}(k) \wedge \text{blank}(k).$

The output of the five methods being compared here is given below.

a) Winston

Winston's algorithm is sensitive to the order in which examples are presented to it. It produces two different concepts for the given three examples of Figure 4.10. The first is the result of presenting examples in the order E_3, E_1, E_2 , and the concept generated can be expressed in first-order logic using the above notation as follows :

$\exists t \exists u \exists v \exists w \exists x \exists y \exists z (\text{size}(x, \text{medium}) \wedge \text{texture}(x, \text{blank}) \wedge \text{shape}(x, \text{polygon}) \wedge \text{ontop}(x, y) \wedge \text{size}(y, w) \wedge \text{texture}(y, u) \wedge \text{size}(z, v) \wedge \text{texture}(z, t)),$ or, in English :

There is a blank, medium polygon on top of another object that has a texture and size. There is also another object with texture and size.

If the examples are presented in the order E_1, E_2, E_3 , then the concept learned is :

$\exists x (\text{size}(x, \text{large}) \wedge \text{texture}(x, \text{blank})),$ or, in English :

There is a large blank object.

b) Hayes-Roth

The algorithm given by Hayes-Roth learns three concepts from the three examples in Figure 4.10. These are given below in first-order logic notation :

$\exists x \exists y (\text{medium}(x) \wedge \text{blank}(x) \wedge \text{ontop}(x, y)),$ or, in English :

There is a blank, medium object on top of another object.

$\exists x \exists y (\text{medium}(x) \wedge \text{blank}(y) \wedge \text{large}(y) \wedge \text{ontop}(x, y)),$ or, in English :

There is a medium object on top of a blank large object.

$\exists x \exists y \exists z (\text{medium}(x) \wedge \text{blank}(x) \wedge \text{large}(y) \wedge \text{blank}(y) \wedge \text{shaded}(z)),$ or, in English :

There is a medium blank object, a large blank object, and a shaded object.

c) Vere

Vere's learning algorithm produces a large number of generalizations for the examples in Figure 4.10. This is due to the fact that many-to-one bindings of variables are permitted, which results in a large number of generalizations. Three of the most specific generalizations are given below :

$\exists x \exists y \exists z \exists w (medium(x) \wedge ontop(x, y) \wedge large(y) \wedge blank(y) \wedge blank(z) \wedge shaded(w))$, or, in English :

There is a medium object on top of a large blank object, a blank object, and a shaded object.

$\exists x_1 \exists x_2 \exists x_3 \exists x_4 (medium(x_1) \wedge blank(x_1) \wedge ontop(x_1, x_2) \wedge shaded(x_3) \wedge large(x_4) \wedge blank(x_4))$, or, in English :

There is a medium, blank object on top of some other object, there is a shaded object and a large blank object which are related in some way.

$\exists x_1 \exists x_2 \exists x_3 \exists x_4 (medium(x_1) \wedge ontop(x_1, x_2) \wedge large(x_2) \wedge blank(x_2) \wedge shaded(x_3) \wedge blank(x_4))$, or, in English :

There is a medium object on top of a large blank object, a shaded object, and a blank object.

d) Dieterich and Michalski

As in the case of Vere's algorithm, Michalski's algorithm produces a large number of generalizations. Some of the more significant ones are :

$\exists x \exists y (ontop(x, y) \wedge size(x, medium) \wedge shape(x, polygon) \wedge texture(x, blank) \wedge (size(y, medium) \vee size(y, large)) \wedge (shape(y, rectangle) \vee shape(y, circle)))$, or, in English :

There exists a medium blank polygon on top of a medium or large rectangle or circle.

$\exists x \exists y (ontop(x, y) \wedge size(x, medium) \wedge (shape(x, circle) \vee shape(x, rectangle) \vee shape(x, square)) \wedge size(y, large) \wedge texture(y, blank) \wedge (shape(y, rectangle) \vee shape(y, box) \vee shape(y, ellipse)))$, or, in English :

There exists a medium circle or rectangle or square on top of a large blank rectangle or box or ellipse.

$\exists x \exists y (ontop(x, y) \wedge size(x, medium) \wedge shape(x, polygon) \wedge (size(y, medium) \vee size(y, large)) \wedge (shape(y, rectangle) \vee shape(y, ellipse) \vee shape(y, circle)))$, or, in English :

There exists a medium polygon on top of a medium or large rectangle or ellipse or circle.

$\exists x ((size(x, small) \vee size(x, medium)) \wedge (shape(x, circle) \vee shape(x, rectangle)) \wedge texture(x, shaded))$, or, in English :

There exists a small or medium shaded circle or rectangle.

e) Our method

We applied our method to all possible combinations of the order of presentation of the three examples (i.e. E_1, E_2, E_3 ; E_2, E_3, E_1 ; and E_1, E_3, E_2). The same

formula was obtained in all three cases. The axioms used for performing resolutions were :

$$\forall x(\text{diamond}(x) \rightarrow \text{polygon}(x))$$

$$\forall x(\text{square}(x) \rightarrow \text{polygon}(x))$$

$$\forall x(\text{triangle}(x) \rightarrow \text{polygon}(x))$$

$$\forall x(\text{rectangle}(x) \rightarrow \text{polygon}(x))$$

$$\forall x(\text{box}(x) \rightarrow \text{polygon}(x)).$$

All possible resolutions were performed between these axioms and the three examples.

The learned concept was :

$\exists x \exists y \exists z (\text{medium}(x) \wedge \text{polygon}(x) \wedge \text{blank}(x) \wedge (\text{ontop}(x, y) \vee \text{ontop}(x, z)) \wedge \text{shaded}(y) \wedge (\text{ontop}(y, z) \vee \text{ontop}(x, z)) \wedge \text{large}(z) \wedge \text{blank}(z))$, or, in English :

There is a medium blank polygon x on top of one of two objects y or z ; y is shaded, and z is a large blank object on top of which is either x or y .

The working of the algorithm for the order of presentation E_1, E_2, E_3 is given in the appendix.

Discussion

The concepts generated by the algorithms of Winston, Hayes-Roth, and Vere are less specific than the concept generated by our algorithm, as can be seen at a glance, since the latter contains more detail than the concepts generated by these three algorithms. Dietterich and Michalski's algorithm generates a number of very different descriptions. While these descriptions do not capture all the detail that our description does, they contain some information that our description does not. The concepts generated by Dietterich and Michalski's algorithm describe at most two objects, whereas ours describes three objects and detects relationships between these objects which none of the algorithms find. The difference in the concepts generated by these two algorithms is due in part to the differences in the description languages used by the two methods.

The great advantage of our representation is that it allows disjunctions and quantifiers to be represented. This automatically makes the scope of application of our algorithm much wider than the first three algorithms studied. Dietterich and Michalski's algorithm does allow internal disjunctions and existential quantifiers, but not universal quantifiers.

In conclusion, we see that our algorithm succeeds in learning a concept from the given three examples which cannot be learned by any of the four algorithms used for comparison in this section. Our algorithm performs better than the first three algorithms in that the concept it learns is more specific than those learned

by these three algorithms. Dietterich and Michalski's algorithm succeeds in finding a number of concepts which are neither more specific nor less specific than the concept generated by our algorithm. However, our algorithm has much wider applicability than theirs, since it can handle arbitrary first-order formulas. Dietterich and Michalski's algorithm uses a special-purpose language, which greatly limits its usefulness in any other field.

5. Mechanizing mathematical induction

5.1 Introduction

Even though theorem provers today are able to prove a vast number of non-trivial mathematical theorems, the proofs of theorems which require the use of mathematical induction present a special problem. A formal statement of the principle of mathematical induction runs as follows : for every predicate P , if $P(m)$ is true for all minimal elements m of a well-founded partial ordering $<$, and if for all y , $P(x)$ being true for all elements x less than y implies that $P(y)$ is true, then $P(x)$ is true for all x . This statement cannot be expressed in first-order logic, since it involves quantification over predicates. However, second-order logic is not even semi-decidable, and there exists no complete deduction system for second-order logic. Recall that first-order logic is semi-decidable but not decidable.

For this reason, due to the properties of second-order logic, many attempts have been made to prove inductive theorems using first-order logic. Given the limitations of first-order logic, it is necessary to provide a theorem prover with the necessary inductive hypotheses which it will need in order to prove a theorem. However, discovering these inductive hypotheses is a non-trivial task. Another problem is to discover a suitable well-founded ordering for the elements of the domain involved. This chapter is devoted to methods for mechanically generating such a well-founded ordering and inductive hypotheses for certain classes of theorems.

Two approaches to generating inductive hypotheses are adopted here. The first relies on our method for deriving logical consequences. The second is based on the fact that there exists a certain class of theorems such that all ground instances of these theorems are provable by first-order methods. Moreover, such proofs can have a similar structure. Thus it is possible to detect which inductive hypotheses are required for a proof of the theorem by induction, by comparing proofs of different ground instances of the theorem. We describe a complete method for proving such theorems. Related work in the past has focused primarily on proving equational theorems using term rewriting techniques. The method described here is more general and is applicable to equational as well as non-equational theorems.

5.2 Related work

One of the hardest problems in discovering an inductive proof is discovering an appropriate application of the principle of induction. Boyer and Moore's prover [Boyer and Moore 79] is perhaps one of the best-known systems which can perform inductive proofs. Their prover uses the definition of a recursive function to suggest an induction scheme. When a recursive function is defined, a measure and a well-founded relation must be provided along with the function definition such that in every recursive call the measure of the variables involved in the recursive call decreases. This guarantees that a function call will not result in an infinite sequence of invocations of the function. This measure and well-founded relation suggest an induction scheme for this function on the variables whose measure decreases with every function call. Boyer and Moore's prover has succeeded in finding inductive proofs of an impressive number of theorems including the unique prime factorization theorem and other number theoretical theorems. It has also been used to prove the correctness of non-trivial programs such as a simple optimizing expression compiler and a fast string searching algorithm.

Another system which makes use of a function definition to generate an induction scheme is described in [Zhang et al. 88]. The idea is similar to that of Boyer and Moore's prover in that different induction schema are used for different functions determined by their definitions. The non-recursive equation(s) in a definition suggests the basis step of an induction proof, while the recursive equation in the definition suggests the inductive step of a proof. The notion of a cover set is introduced, which is a finite set of terms covering all the elements of the constructor model; in other words, a cover set is a finite set of terms which "describes" every ground constructor term of that sort. One of the hardest problems in applying the cover set induction principle is to find a suitable cover set. As in [Boyer and Moore 79], the function symbols in the conjecture and their definitions offer an insight into the problem. The method described is a generalized version of the structural induction principle and is as powerful as Boyer and Moore's induction method; however, the experiments performed with this system are still very limited as compared to the achievements of Boyer and Moore's prover.

In a preliminary report, Biundo et al. [Biundo et al. 86] describe a system in which mathematical induction is being incorporated. Given a finite set of axioms and a formula ϕ , their induction theorem prover first attempts to prove ϕ without using induction. If this attempt fails, an induction formula is generated from ϕ , after simplifying ϕ , according to Aubin's method [Aubin 79a], [Aubin 79b]. It may happen that a given formula is not directly provable by induction, i.e. the induction formula obtained from the initial one provides an induction hypothesis which is too

weak to be usable in the induction step. In such cases it is necessary to find a more general formula which is sufficient for the initial one such that the induction hypothesis "carries" the induction. Several techniques for generalizing formulas are being studied for this purpose.

A lot of work has been done in the field of "inductionless induction", which is basically proving theorems which would normally require inductive proofs by using term rewriting techniques. Musser [Musser 80] discovered the use of the Knuth-Bendix completion procedure [Knuth and Bendix 70] for proving equations by induction from an equational specification of data types. Since the classical induction principle is not explicitly invoked in this method, it has been called the inductionless induction method. The general idea is that an equation is valid in the initial algebra if and only if adding it to the set of axioms does not result in an inconsistency. The pioneering work of Musser led to the development of this method for proving inductive properties of data types. The equational axioms of an algebraic specification of a data type can often be formed into a convergent set of rewrite rules. If one adds a rewrite rule corresponding to a data type property whose proof requires induction, convergence may be destroyed, but often can be restored by using the Knuth-Bendix algorithm to generate additional rules. A convergent set of such rules can be used as a decision procedure for the equational theory for the axioms plus the property added. This fact, combined with a "full specification" property of axiomatizations, leads to a new method of proof of inductive properties, not requiring the explicit use of an inductive rule of inference. The significance of "full specification" lies primarily in that if one has a collection of types that has been shown to be fully specified, and one extends the collection with a new type specification so that the augmented collection is also fully specified, then the added specification does not introduce any new constants into the old types. Thus to attempt to prove that an equation $\alpha = \beta$ is in the inductive theory of the collection of types, one adds a new rewrite rule (either $\alpha \rightarrow \beta$ or $\beta \rightarrow \alpha$ according to the finite termination criterion being used) and performs the Knuth-Bendix algorithm. There are three possible outcomes : (1) The algorithm terminates after generating a finite number of additional rules, none of which is *true* \rightarrow *false*, with the convergence property affirmed. This means that $\alpha = \beta$ is a theorem; (2) The rule *true* \rightarrow *false* is generated, which means that $\alpha = \beta$ is not a theorem; or (3) The Knuth-Bendix algorithm does not terminate, in which case no definite information is gained about whether $\alpha = \beta$ is a theorem or not. The main limitation here is the difficulty of proving the finite termination property of the rules generated (finite termination of arbitrary sets of rewrite rules is undecidable). Another issue is that of the practicality of meeting the requirement of "full specifications".

Goguen [Goguen 80] describes a more general approach to inductionless induction than Musser. He proves the correctness of algebraic methods for deciding the equivalence of expressions by applying rewrite rules, and for proving inductive equational hypotheses without using induction. He also shows that the equations true in the initial algebra are just those provable by structural induction. The results generalize and rigorize Musser's method for proving inductive hypotheses using the Knuth-Bendix algorithm, by showing that under certain conditions, an equation is true if and only if it is consistent.

Toyama [Toyama 86] proposes a method for testing equivalence in a restricted domain of two given term rewriting systems. By using the Church-Rosser property and the reachability of term rewriting systems, the method can be used to prove equivalence of these systems without the explicit use of induction. The method proposed is an extension of inductionless induction methods developed in [Musser 80], [Goguen 80], [Huet and Hullot 82], etc. and allows the extension of inductionless induction not only to term rewriting systems with the termination property but also various reduction systems. Toyama proves certain theorems about abstract reduction systems, which are then applied to term rewriting systems to prove equivalence in a restricted domain of two term rewriting systems. His method has a wider applicability than the inductionless induction methods proposed by Musser and others, since their method requires the strongly normalizing property to hold, which is very restrictive and not true of many term rewriting systems.

McCarthy [McCarthy 70] describes a method called recursion induction for proving the equivalence of recursively defined functions. The method works as follows : suppose a function f is defined over a set A , and suppose g and h are two other functions with the same domain as f and which are defined for all elements of A . Suppose further that g and h satisfy the equation which defined f . Then the values of g and h agree for all elements of A . This method of proving functions g and h equivalent is called recursion induction. Some elementary results in the elementary theory of numbers and in the elementary theory of symbolic expressions are provable using recursion induction. In number theory one gets as far as the theorem that if a prime p divides ab , then it divides either a or b . However, to formulate the unique factorization theorem requires a notation for dealing with sets of integers. One of the most immediate problems in extending this theory is to develop better techniques for proving that a recursively defined function converges.

The concept of structural induction can be explained as follows : to prove that some property holds for some inductively defined data structure, we show that it holds for the most elementary data, and that it will hold for data of any degree of complexity provided that it holds for all data of lesser complexity. We may

then deduce that it holds for all data [Burstall 69]. This is a special case of a more general rule termed Noetherian induction : “Let A be an ordered set with minimum condition and B a subset of A which contains any element $a \in A$ whenever it contains all the elements $x \in A$ such that $x < a$. Then $B = A$.” (A set A satisfies the minimum condition if every non-empty subset of A has a minimal element.) The structural induction principle differs from the usual “course of values” induction in two ways : it allows for partial ordering, instead of total ordering, and it allows induction over the transfinite steps. Examples of some proofs by structural induction can be found in [Burstall 69].

In [Wegbreit and Spitzen 76], the authors introduce a method of proof called generator induction, used for proving properties of programs, which may be stated informally as follows : to prove that all instances of a class C have some property P , prove that (1) all instances have the property when they are first created, and (2) all operations F , which may change the value of a class instance, preserve the truth of P . This definition is similar to that of computation induction given in [Manna et al. 73]. The generator induction principle is used to prove certain properties of a hashtable program in Simula. The most important property of generator induction, according to the authors, is that it partitions the program into loosely coupled parts, proves simple properties of the parts, and demonstrates that the parts are composed according to simple rules. This allows the decomposition of a proof into small, comprehensible units corresponding to the structure of the program.

5.3 Description of the first method

We give below a brief overview of the first approach we will be using for deriving proofs of theorems. Consider what is provable by induction, where all induction hypotheses are expressible in first-order logic and all orderings are known. This gives a precisely defined class of formulas. Given a theorem T to be proved, we first try to prove it without using induction, using a resolution theorem prover. If this attempt fails, we try to prove the theorem using induction. The first problem to be tackled is to find a suitable induction scheme, i.e. we must discover a suitable well-founded ordering to be used in the application of the principle of induction. Once this has been done, an attempt is made to prove the theorem using this ordering and the induction principle. However, it may happen that this proof fails too, since this theorem may itself depend on another inductive hypothesis or lemma A . Then we have

$$\text{AXIOMS} \wedge A \rightarrow T$$

where “AXIOMS” is the set of axioms required for the proof of this theorem. Therefore

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A$$

i.e. $\neg A$ is a logical consequence of $\text{AXIOMS} \wedge \neg T$. We can therefore use our method for generating logical consequences and unskolemization to derive $\neg A$ from $\text{AXIOMS} \wedge \neg T$. This is a support strategy and will not generate all possible inductive theorems from the axioms, since it makes use of the negation of the theorem T as well as the set of axioms to generate inductive hypotheses. This method can be extended to theorems which depend on more than one inductive hypothesis. The remainder of this section elaborates the ideas outlined above.

5.3.1 Discovering a well-founded ordering

Recall that we have assumed that all well-founded orderings are known (these could be partial as well as total orderings). Now suppose that we are trying to prove the theorem

$$\forall x A(x)$$

where x ranges over some domain D . Consider the set of formulas of the form $A(t)$, where t is a ground term belonging to D and $A(t)$ is first-order provable. We prove some subset of these formulas one by one, noting the proof times for each formula. We denote the time taken to prove $A(t)$ by $PT(t)$. This suggests an ordering in that objects which are smaller in the ordering will probably have smaller proof times. We therefore pick an ordering “ \succ ” such that

$$(X \succ Y) \rightarrow (PT(X) > PT(Y))$$

(at least most of the time).

Example 5.1 Consider the following theorem to be proved by induction :

$$\forall X(\text{reverse}(\text{reverse}(X)) = X),$$

where “*reverse*” is the usual function which reverses lists, and where X ranges over the set of all lists. We first try to prove the theorem for some ground terms using the theorem prover OTTER, a resolution theorem prover developed at the Argonne National Laboratory [McCune 89]. We observe the following proof times for the ground terms given below :

Theorems proved	Time taken (seconds)
$\text{reverse}(\text{reverse}([\])) = [\]$	0.26
$\text{reverse}(\text{reverse}([1])) = [1]$	0.34
$\text{reverse}(\text{reverse}([a])) = [a]$	0.34

$reverse(reverse([2, 1])) = [2, 1]$	1.00
$reverse(reverse([a, b])) = [a, b]$	1.00
$reverse(reverse([3, 2, 1])) = [3, 2, 1]$	2.16
$reverse(reverse([5, 2, 9])) = [5, 2, 9]$	2.16

From the above proof times, the following observations can be drawn :

1. The proof times increase as the length of the list being substituted for X increases; here we have

$$PT([]) < PT([1]) = PT([a]) < PT([2,1]) = PT([a, b]) < PT([3,2,1]) \dots$$

2. The proof times are identical for different ground terms which are lists of the same length; here we have

$$PT([1]) = PT([a]), PT([2,1]) = PT([a, b]), \text{ and so on.}$$

A well-founded ordering \succ which satisfies the condition

$$X \succ Y \rightarrow PT(X) > PT(Y)$$

is therefore the ordering which is defined as follows : for lists X and Y , $X \succ Y$ if and only if the length of list X is greater than the length of list Y ; $X = Y$ if and only if the length of list X is equal to the length of list Y ; and $X \prec Y$ if and only if $Y \succ X$. The minimal element in this ordering is the empty list $[]$. •

5.3.2 Using the induction principle

Once a well-founded ordering W has been found using the method described in the previous section, we must now apply the principle of induction to prove the given theorem $T = \forall x A(x)$. This is done as follows :

Base case : We prove

$$A(m)$$

for all minimal elements m of the ordering W by negating $A(m)$, adding it to the set of axioms, and using resolution to derive the empty clause. If this procedure fails, this means that some induction hypothesis (hypotheses) is (are) required to prove the base case. Sections 5.3.3 – 5.3.4 explain how this case is dealt with.

When the proof of the base case is obtained, we proceed to the induction step.

Induction step : Let $pred(X)$ be the set of elements which precede X in the ordering W . Then we need to prove

$$\forall X \left(\bigwedge_{Y \in pred(X)} A(Y) \rightarrow A(X) \right).$$

Again, this formula is negated, added to the set of axioms and resolution is used to try and derive the empty clause. If the empty clause cannot be thus derived, then as before, one or more induction hypotheses may be required to obtain the proof. This is dealt with in the next two sections.

If a proof is obtained by resolution alone, then the theorem is proved by induction.

Example 5.2 Consider the following theorem :

$$\forall Y \forall Z (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y)))$$

The list notation used here is the same as that used in the programming language LISP. In particular, if X is an element and Y is a list, then $\text{cons}(X, Y)$ is a list of length one more than the length of Y , and contains the element X followed by the list Y . $[]$ or “*nil*” represents the empty list.

Let us try to prove the theorem by induction on Y . Using the same method as in the previous section, we can establish that the same well-founded ordering as that of Example 5.1 can be used here (i.e. for lists X and Y , $X \succ Y$ if and only if the length of list X is greater than the length of list Y ; $X = Y$ if and only if the length of list X is equal to the length of list Y ; and $X \prec Y$ if and only if $Y \succ X$). The minimal element in this ordering is the empty list $[]$, therefore here the base case is :

Base case : $\forall Z (\text{reverse}(\text{append}([], \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}([])))$.

This was proved by OTTER in 0.28 seconds (see the appendix for proof).

Induction step : An element preceding X in the given ordering could be any element of length one less than X . In particular, the list $\text{cdr}(X)$, where $\text{cdr}(X)$ represents the same list as X with the first element removed, is a list with one element less than X . Thus we now need to prove that

$$\forall Y \forall Z ((\text{reverse}(\text{append}(\text{cdr}(Y), \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(\text{cdr}(Y)))) \rightarrow (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y))))$$

This was proved by OTTER in 1.86 seconds (see the appendix for proof).•

5.3.3 Finding one induction hypothesis

Sometimes it may happen that either the base case or the inductive step (or both) of the previous section cannot be proved by resolution alone, i.e. by first-order methods alone. This can occur, for example, if the proof depends on some other lemma which itself needs to be proved by induction before a proof for the actual theorem being proved can be found. Suppose that the proof depends on one lemma A , i.e.

$$\text{AXIOMS} \wedge A \rightarrow T$$

where “AXIOMS” is the set of axioms for the theorem T being proved, and where T can be proved from AXIOMS \wedge A using first-order reasoning. This implication can be rewritten as

$$\neg \text{AXIOMS} \vee \neg A \vee T$$

which is the same as

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A.$$

$\neg A$ can therefore be derived from $(\text{AXIOMS} \wedge \neg T)$ using resolution. To do this, we need to Skolemize $\text{AXIOMS} \wedge \neg T$ before performing resolutions among clauses thus obtained. The clauses which represent $\text{Sk}(\neg A)$ can thus be derived by resolution from $\text{Sk}(\text{AXIOMS} \wedge \neg T)$. We then unskolemize and negate $\text{Sk}(\neg A)$ to obtain the lemma A, using our unskolemization algorithm. Lemma A can now be proved using the methods of Sections 5.3.1 – 5.3.2. Since lemma A has been shown to be valid, and since $\text{AXIOMS} \wedge A \rightarrow T$, the theorem T is also valid, and a proof for T has thus been found.

Example 5.3 We continue with the proof of the theorem of Example 5.1, which was

$$\forall X(\text{reverse}(\text{reverse}(X)) = X).$$

Recall that we found a well-founded ordering W for this theorem in Example 5.1 whose minimal element was the empty list $[\]$. The base step for the proof therefore consists of proving

$$\text{reverse}(\text{reverse}([\])) = [\],$$

which was already done in Example 5.1.

Induction step : An element preceding X in the ordering W could be any element of length one less than X . In particular, the list $\text{cdr}(X)$, where $\text{cdr}(X)$ represents the same list as X with the first element removed, is a list with one element less than X . Thus we need to prove that

$$\forall X((\text{reverse}(\text{reverse}(\text{cdr}(X))) = \text{cdr}(X)) \rightarrow (\text{reverse}(\text{reverse}(X)))).$$

(Call this result T' .) We try to derive the empty clause from $\text{AXIOMS} \wedge \neg T'$, but since T' cannot be proved by induction alone, this attempt fails. However, we succeed in deriving the following clause by resolution from $\text{AXIOMS} \wedge \neg T'$:

$$\forall X(\text{reverse}(\text{append}(\text{reverse}(\text{cdr}(X)), \text{cons}(\text{car}(X), [\]))) \neq \text{cons}(\text{car}(X), \text{reverse}(\text{reverse}(\text{cdr}(X)))).$$

We unskolemize this clause by replacing $\text{reverse}(\text{cdr}(X))$ and $\text{car}(X)$ by new existential variables Y and Z respectively. This yields the formula

$$\exists Y \exists Z(\text{reverse}(\text{append}(Y, \text{cons}(Z, [\]))) \neq \text{cons}(Z, \text{reverse}(Y))).$$

Negating, we obtain the following

Lemma A = $\forall Y \forall Z (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y)))$.

Lemma A was proved by induction (without the use of any lemmas) in Example 5.2, using the methods of Sections 5.3.1 – 5.3.2. Hence the proof of our theorem

$$\forall X (\text{reverse}(\text{reverse}(X)) = X)$$

is complete. •

5.3.4 Finding more than one induction hypothesis

It may happen that the proof of a theorem T depends on more than one inductive lemma. For example, suppose that inductive lemmas A and B are needed to prove T ; in other words, T is first-order derivable from $\text{AXIOMS} \wedge A \wedge B$. Then as before, since

$$\text{AXIOMS} \wedge A \wedge B \rightarrow T,$$

we therefore have

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A \vee \neg B.$$

Therefore from $\text{AXIOMS} \wedge \neg T$, we can derive $\neg A \vee \neg B$ by first-order reasoning. This can be done by resolution from $\text{Sk}(\text{AXIOMS} \wedge \neg T)$. The clauses obtained from these resolutions representing $\text{Sk}(\neg A \vee \neg B)$ can then be unskolemized and negated (as we did in Section 5.3.3 for lemma A). We thus obtain $A \wedge B$, and each of lemma A and lemma B can be proved by induction using the methods of Sections 5.3.1 – 5.3.2. This method can be extended to any number of inductive lemmas; however, the method rapidly becomes more and more complicated as the number of lemmas increases. For this reason, it may be preferable to use our second approach, described in the next section, for generating inductive hypotheses for such theorems.

5.4 Description of the second method

Suppose that we are trying to prove some theorem T , and suppose that we fail to find a proof of the theorem using standard first-order methods. This suggests that induction may be required to prove the theorem.

If induction is to be used, the first problem is to find a well-founded ordering for the elements. This can be done by the methods described in Section 5.3.1 and this problem will not be further dwelt upon here. If the theorem can now be proved by induction using the well-founded ordering thus discovered, then we are done; if not, then the proof of the theorem may require one or more lemmas, which themselves need to be proved by induction. Our concern in this section is to discover what these lemmas are. If only one lemma is required, then the method of Section 5.3.3 may

prove useful in finding this lemma; however, the following method can be applied to any theorem whose proof requires one or more lemmas, each of which have to be proved by induction. By lemma we mean either some theorem which needs to be proved separately or some instance $T(\bar{Y})$ of the theorem $T(\bar{X})$, where $\bar{Y} < \bar{X}$; such lemmas are also called inductive hypotheses.

Suppose the proof of a theorem T , for which a well-founded ordering P has been discovered, requires n lemmas A_1, A_2, \dots, A_n (these are all unknown). Suppose that T contains m variables ($m \geq 1$), where the i^{th} variable is drawn from some domain D_i for each i , $1 \leq i \leq m$. Let \bar{X} be an m -tuple consisting of these m variables. We write the theorem T as $T(\bar{X})$. Then a proof of the theorem $T(\bar{X})$ will proceed according to the following two steps :

1. Proof of the base case :

The theorem $T(\bar{X})$ is proved to be true for the minimal elements of the ordering P . We will show below that $T(\bar{m})$, for minimal elements \bar{m} of the ordering P , is first-order provable for a certain class of theorems T .

2. Inductive step :

Now the inductive step of the theorem $T(\bar{X})$ is proved. Using lemmas A_1 through A_n as axioms, a proof of $T(\bar{X})$ can be obtained by resolution.

This concludes a proof of $T(\bar{X})$ by induction, using the n lemmas A_1 through A_n as axioms.

Now consider the proof of $T(\bar{Y})$ for some ground element $\bar{Y} \in D_1 \times D_2 \times \dots \times D_m$. This proof can be performed by performing exactly the same steps as in 2 above, except that we now have none of the lemmas A_1 through A_n . As a result of this, ground instances of these lemmas will have to be proved. We will show below that all these ground instances are first-order provable for a certain class of theorems. Thus in this proof of $T(\bar{Y})$, we can find subproofs of lemmas $A_1(\bar{Y})$ through $A_n(\bar{Y})$ ($A_i(\bar{Y})$ denotes the lemma A_i with variables in A_i instantiated to the corresponding values in \bar{Y}).

Since this is true for all ground elements \bar{Y} , the above can be repeated for ground elements $\bar{Y}_1, \bar{Y}_2, \bar{Y}_3, \dots$, and so on. In each proof of $T(\bar{Y}_i)$, we can find subproofs of lemmas $A_1(\bar{Y}_i), A_2(\bar{Y}_i), \dots, A_n(\bar{Y}_i)$.

Now we compare the proofs of $T(\bar{Y})$ for different ground \bar{Y} . These proofs will be similar in structure except that different instances of the lemmas A_1 through A_n will appear in these proofs. By detecting these different instances, we should be able to reconstruct the n lemmas A_1 through A_n . Once these lemmas are known, the theorem $T(\bar{X})$ can be proved by induction using the well-founded ordering P .

In the following theorem, we will show that this method is complete for theorems which can be proved using the usual induction principle, subject to the fol-

lowing restriction. In order to show that this method is complete, we will need to be able to tell whether certain ground terms t which appear in proofs are less than a given ground term z or not. Since ground terms may contain Skolem functions or other functions, it may not always be possible to deduce whether $t < z$ or not. Thus we will only allow those functions t, z for which it is possible to tell whether $t < z$ is true or not. For example, if $z = 3$ and $t = plus(1, 1)$ (where "plus" is the usual addition function for natural numbers), it is possible to deduce that t is less than z ; however, if $z = 3$ and $t = f(5)$ for some Skolem function f , then we cannot tell whether $t < z$ or not.

The feasibility of this method is established in the following theorem.

Theorem 5.1 The method suggested above is complete for theorems that can be proved by first-order logic with the following induction principle:

$$(\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)))$$

where $<$ is a well-founded ordering. Additionally, none of the clauses used should contain terms containing Skolem symbols or other functions for which the question of whether any of these is less than another term is undecidable.

Proof : Suppose we want to prove $P(z)$ for some ground z , where $\forall xP(x)$ is a theorem which can be proved by first-order logic with the above induction principle.

We know that $(\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)))$ is true. Express this as

$$(\forall x(\exists y)[(y < x \rightarrow P(y)) \rightarrow P(x)]).$$

Substitute the ground term z for x to get

$$(\exists y)[(y < z \rightarrow P(y)) \rightarrow P(z)].$$

Then there are finitely many y_i such that

$$[(y_0 < z \rightarrow P(y_0)) \rightarrow P(z)] \vee \dots \vee [(y_n < z \rightarrow P(y_n)) \rightarrow P(z)]$$

is true. To see this, note that if $(\exists x)A(x)$ is valid for any first order formula $A(x)$, we know there exist finitely many terms t_i such that $A(t_1) \vee \dots \vee A(t_n)$ is valid. We can show this by converting $\neg(\exists x)A(x)$ to clause form with a new predicate for $A(x)$, and looking at the instances of this predicate used in the derivation of the empty clause by resolution.

The above formula can be rewritten as

$$[\neg(y_0 < z \rightarrow P(y_0)) \vee P(z)] \vee \dots \vee [\neg(y_n < z \rightarrow P(y_n)) \vee P(z)]$$

$$\text{i.e. } [\neg(y_0 < z \rightarrow P(y_0)) \vee \dots \vee \neg(y_n < z \rightarrow P(y_n))] \vee P(z)$$

$$\text{i.e. } \neg[(y_0 < z \rightarrow P(y_0)) \wedge \dots \wedge (y_n < z \rightarrow P(y_n))] \vee P(z)$$

$$\text{i.e. } [(y_0 < z \rightarrow P(y_0)) \wedge \dots \wedge (y_n < z \rightarrow P(y_n))] \rightarrow P(z).$$

Now consider any conjunct $(y_i < z \rightarrow P(y_i))$ in the conjunction on the left side of the above implication. If $y_i < z$ is true, then this conjunct is equivalent to

$P(y_i)$; if $y_i < z$ is false, then this conjunct is equivalent to *true* and can therefore be omitted from the above conjunction. We are therefore interested in knowing whether $y_i < z$ is true or false for every i , for $1 \leq i \leq n$. From the assumption in the theorem statement, this question can be answered for all y_i .

Let T be the set of all y_k 's in $\{y_1, y_2, \dots, y_n\}$ such that $y_k < z$. Then

$$\bigwedge_{y \in T} P(y) \rightarrow P(z)$$

Thus $P(z)$ is derivable from a finite conjunction $P(y_{j_1}) \wedge \dots \wedge P(y_{j_q})$ by first-order logic methods, where all the y_{j_i} 's are ground elements less than z .

Repeating the above argument for each of the elements of the set T , we will eventually get

$$P(m_1) \wedge \dots \wedge P(m_s) \rightarrow P(z)$$

where all the m_i 's are minimal elements of the well-founded ordering $<$ (this follows from the fact that $<$ is a well-founded ordering).

Now consider the proof of $P(m_i)$ for some minimal element m_i ($1 \leq i \leq s$). Since m_i is a minimal element, $P(m_i)$ is provable from the given axioms and given lemmas. If the proof of $P(m_i)$ requires the use of lemmas proved by induction previously, then by a simple induction argument on the size of the proof, using the same method as above, we see that we will eventually obtain a first-order proof of $P(m_i)$ from the axioms.

Thus we see that $P(z)$ can be proved by first-order methods; also, this proof is made up of proofs of some $P(y)$'s for $y < z$, which in turn are made up of proofs of some $P(w)$'s for $w < y$, and so on. In other words, each of these proofs have a similar structure, and the theorem is proved. •

A similar theorem can be proved for a slightly different version of the induction principle. In this theorem, no assumptions need to be made regarding the decidability of whether one term is less than another.

Theorem 5.2 The method suggested above is complete for theorems that can be proved by first-order logic with the following induction principle:

If $<$ is a well-founded ordering, and if for all x , $P(x)$ can be proved by first-order logic from the infinite conjunction of $P(y)$ for all $y < x$, then for all x , $P(x)$ is true.

Proof : Suppose we want to prove $P(y)$ for some ground y , where $\forall x P(x)$ is a theorem which can be proved by first-order logic with the above induction principle.

From the assumption in the theorem statement, we know that $P(y)$ can be proved by first-order logic methods from

$$P(y_1) \wedge P(y_2) \wedge \dots \wedge P(y_n) \wedge \dots$$

where $y_i < y \forall i \geq 1$, i.e. $\bigwedge_{i=1}^{\infty} P(y_i) \rightarrow P(y)$.

Therefore by the compactness principle, there exists a finite subset $\{y_{j_1}, y_{j_2}, \dots, y_{j_m}\}$ of the y_i 's such that

$$P(y_{j_1}) \wedge P(y_{j_2}) \wedge \dots \wedge P(y_{j_m}) \rightarrow P(y).$$

Repeating the above argument for each of $P(y_{j_1})$ through $P(y_{j_m})$ in place of $P(y)$, we will eventually get

$$P(m_1) \wedge \dots \wedge P(m_r) \rightarrow P(y)$$

where all the m_i 's are minimal elements of the well-founded ordering $<$ (this follows from the fact that $<$ is a well-founded ordering). As in the proof of Theorem 5.1, since m_i is a minimal element, it is provable from the given axioms and given lemmas. If the proof of $P(m_i)$ requires the use of lemmas proved by induction previously, then by a simple induction argument on the size of the proof, using the same method as above, we see that we will eventually obtain a first-order proof of $P(m_i)$ from the axioms.

Therefore $P(y)$ is also first-order provable. Also, the proof of $P(y)$ can be constructed from the proofs of $P(y_{j_1})$ through $P(y_{j_m})$, each of which in turn can be constructed from a finite conjunction of $P(z_k)$'s, for $z_k < y_{j_k}$ ($1 \leq k \leq m$), and so on. Thus each of these proofs have a similar structure, and the theorem is proved. •

Limitations of this method

1. The first point to note is that for any ground element y , not all proofs of $P(y)$ will have a similar structure to $P(y')$ for other ground elements y' . Potentially, there may exist a large number of different proofs of each ground instance. However, there does exist at least one such proof, as demonstrated in the preceding theorems. We will need to search through the proofs to find one such proof.
2. Given proofs of ground instances of $P(x)$, it is a non-trivial task to detect the similarity in structure between these proofs.

Examples illustrating the use of this method can be found in the appendix.

5.5 Comparison with other methods

The first method given in this chapter uses a support strategy to generate inductive hypotheses from the axioms; the second method is more general and makes use of structural similarities in the proofs of ground instances of the theorem being proved to discover suitable inductive hypotheses. Much of the work in the field

of mechanizing mathematical induction is concentrated in the field of inductionless induction, described in more detail in Section 5.2. As mentioned earlier, inductionless induction applies term rewriting techniques for proving equational theorems. Our methods are more general than these since they can be applied to any theorem which can be proved using the principle of induction. Some more work needs to be done in order to improve the efficiency of our method.

6. Conclusion

6.1 Summary

In this dissertation, we have explored a number of different topics. We first saw how a class of logical consequences of first-order formulas can be derived using resolution and unskolemization. We extended the meaning of “unskolemization” to include replacement of some non-Skolem as well as Skolem functions by existentially quantified variables. This allowed a larger class of logical consequences to be derived, since certain logical consequences of formulas cannot be derived without unskolemization. A detailed algorithm was given to perform this unskolemization, and the properties of formulas derived by applying the algorithm were described.

The remainder of this dissertation revolved around different applications for the above method for deriving logical consequences. We first used the method as part of an algorithm for the automatic generation of loop invariants. The method was applicable since a loop invariant is a logical consequence of the various conditions which are true each time the loop is traversed. We described methods of directing the search for a valid loop invariant and demonstrated their effectiveness with several examples. The algorithm for generating loop invariants in first-order logic was proved to be sound and complete. This is in contrast to all known methods so far, which are heuristics and are by no means complete.

The next topic discussed was machine learning from examples. Given two examples $E1$ and $E2$, a concept learned from $E1$ and $E2$ is a logical consequence of $E1$ and $E2$. Thus we applied our resolution and unskolemization method for deriving logical consequences to this problem. A graph-based algorithm for learning by extracting common features from examples was described, and the properties of the concepts which can be thus learned were discussed. Applications of this learning algorithm to traditional areas such as the blocks world, as well as the mechanical derivation of loop invariants, were demonstrated. The performance and working of our algorithm was compared with those of four other algorithms from the literature, and it was shown that the performance of our algorithm compared favorably with the other four. This work is significant because none of the learning algorithms so

far have used full first-order logic as their representation language. This greatly widens the scope of applicability of our method.

Finally, we described methods for discovering inductive hypotheses for theorems to be proved by induction. Since the principle of mathematical induction is not expressible in first-order logic, in order to be able to prove theorems by induction using only first-order logic, we need to know which inductive hypotheses will be required for the proofs of the theorems. We saw that certain inductive hypotheses can be generated from the axioms and the negation of the theorem by using our method for generating logical consequences. Another method, which involved extracting inductive hypotheses from proofs of ground instances of the theorem, was described. This method was based on the fact that proofs of ground instances of the theorem can have similar structures, and information about which inductive hypotheses are required can be deduced by comparing the structures of these proofs. The method was shown to be complete for certain classes of theorems. It is more general than a large number of existing methods, since it can be applied to equational as well as non-equational theorems. Much of the existing work on this subject deals only with equational theorems.

6.2 Extensions

6.2.1 Automatic generation of loop invariants

We have developed a novel method of automatically deriving loop invariants for flowchart programs. The methods described in this dissertation have not been actually implemented, but have been manually applied to many examples. Many people have voiced the opinion that the goal of automating the derivation of loop invariants is unattainable (see for example [Dijkstra 85]). Of course, they can be proved wrong only if the method we have developed can be made “acceptably” efficient by the use of suitable strategies. Basically, the function GET-APPROX needs to be implemented with the use of strategies which will include rewriting terms to some normal form to improve the efficiency of the resolution procedure, detecting structural similarities among terms, and so on. The function, as it stands now, provides some guidance to the process of deriving the invariants. Its efficiency can probably be greatly improved with the use of some good heuristics. Owing to the existence of a large number of such heuristics in the literature, this aspect has not been explored in much detail here. However, even though heuristics will be able to improve the performance of our algorithm, the algorithm still stands out from the previous purely heuristic methods in the literature. This is because in our

method, heuristics can be embedded within the framework of a complete and sound algorithm. Thus even if all heuristics fail, our algorithm can still derive a correct loop invariant. This is in direct contrast to previously developed methods, which have not been complete in any sense.

Another issue here is that if the given algorithm fails to return a loop invariant for a given program loop, this could be due to one of two reasons : either the invariant is not expressible given the theory axiomatized, or the program is not correct. These two cases cannot be distinguished at present.

6.2.2 Learning from examples

We propose some extensions and modifications to the learning algorithm presented in Chapter 4.

Allowing many-to-many mappings

The algorithm, as presented in Section 4.4.2, performs a one-to-one mapping of arguments from the two given examples. This corresponds to the notion that distinct objects in the two given examples are represented by distinct variables. However, in certain situations it may be desirable to allow different variables to represent the same object. In such a case, it is necessary to allow many-to-many mappings in the argument graph produced by the algorithm. The choice of whether to consider all possible mappings or a limited number of these can be left to the user. A very minor modification to the learning algorithm will allow this feature to be incorporated into the algorithm.

Allowing a limited number of disjunctions

The learning algorithm at present does not allow disjunctions of clauses to be performed if the clauses have no common predicates. If such disjunctions are necessary, they can be permitted, either without restriction or with a limit on the number of disjunctions allowed. This alteration can easily be built into the algorithm.

Using the algorithm for descriptive generalization

The given algorithm provides a method of deriving a formula EX from two given formulas $E1$ and $E2$ such that $E1 \rightarrow EX$, $E2 \rightarrow EX$. However, note that we could also use the algorithm for deriving a formula EX such that $EX \rightarrow E1$,

$EX \rightarrow E2$. To see this, suppose that we are given formulas $E1$ and $E2$; then apply the algorithm to the formulas $\neg E1$ and $\neg E2$. The algorithm produces a formula E such that $\neg E1 \rightarrow E$ and $\neg E2 \rightarrow E$; taking contrapositives, we get $\neg E \rightarrow E1$, $\neg E \rightarrow E2$. Setting $EX = \neg E$, the result follows. In the terminology of Michalski [Michalski 83], this process is known as *descriptive generalization* and is concerned with establishing new concepts or theories characterizing given facts. In this case E_1 and E_2 are the given facts, and EX is the new concept or theory which is established. This method of inference is also known as *abduction* or *abductive inference* [Patterson 90].

6.2.3 Mechanizing mathematical induction

The methods developed in Chapter 5 for generating inductive hypotheses are complete for certain classes of theorems; they need to be made more efficient by the use of suitable strategies. More research needs to be done into ways of detecting structural similarities among proofs of different ground instances of theorems.

Appendix

Time complexity analysis of the learning algorithm

We analyze the algorithm step by step. For convenience, the main body of the algorithm is listed below again, with the steps numbered :

Algorithm LEARN($E_1, E_2, AXIOMS$)

begin

1. Choose $X_c \in Res(E_1 \wedge AXIOMS)$;
2. Choose $Y_c \in Res(E_2 \wedge AXIOMS)$;
3. Rename the variables in all the clauses of X_c and Y_c so that no two clauses have any variable in common;
4. `build_clause_graph`(X_c, Y_c, E_c);
5. `build_argument_graph`(X_a, Y_a, E_a);
6. `augment_graphs_X`(X_a, E_a, X_c, E_c);
7. `augment_graphs_Y`(Y_a, E_a, Y_c, E_c);
8. `maximum_weight_matching`(M_a, X_a, Y_a, E_a);
9. $M_c := \{(C_1, C_2) \in E_c \mid \text{the } n^{\text{th}} \text{ argument } \alpha_n \text{ of some literal of } C_1 \text{ contains } a \text{ as a subterm and the } n^{\text{th}} \text{ argument } \beta_n \text{ of some literal of } C_2 \text{ contains } b \text{ as a subterm in the same position as } a \text{ appears in } \alpha_n \text{ and } (a, b) \in M_a, \text{ for some positive integer } n, \text{ where these two literals have the same predicate}\}$;
10. For every edge $(a, b) \in M_a$ do
 - if $(a$ and b are distinct) and $(a$ and b are not both variables) then
 - replace unmarked occurrences of a and b in M_c by $Z \leftarrow a$ and $Z \leftarrow b$ (respectively) (Z is a new variable);
 - if $(a$ and b are both variables) then
 - unify all occurrences of a and b in M_c ;
11. $EX := \{C_1 \cup C_2 \mid (C_1, C_2) \in M_c\}$;
12. if $EX = \emptyset$ then $EX := true$;
13. for every Skolem function α in EX do
 - if α is not marked then

replace all occurrences of α in all literals of EX by $X \leftarrow \alpha$, where X is a new variable not occurring elsewhere in any clause;

14. Perform Steps 4 through 7 of the unskolemization algorithm for EX ;

end.

We will not analyze the complexity of performing resolutions, since this is a nondeterministic process. The analysis of the algorithm thus begins with step 3 above. The following symbols are used during this analysis :

$|X_c|$: number of elements in the set X_c

$|Y_c|$: number of elements in the set Y_c

$|X_a|$: number of elements in the set X_a

$|Y_a|$: number of elements in the set Y_a

$|C_1|$: maximum cardinality of a clause in X_c

$|C_2|$: maximum cardinality of a clause in Y_c

arg : maximum length of an argument of a clause in $X_c \cup Y_c$ (i.e. maximum number of symbols in an argument; e.g. $f(x, g(y))$ has 4 symbols, viz. f, x, g, y)

$arity$: maximum arity of a predicate in a clause in $X_c \cup Y_c$

$|E_c|$: number of edges of the clause graph (bounded above by $(|X_c| + |Y_c|)^2$)

$|E_a|$: number of edges of the argument graph (bounded above by $(|X_a| + |Y_a|)^2$)

$|M_a|$: size of a maximum weight matching of the argument graph (bounded above by $max(|X_a|, |Y_a|)$)

$|M_c|$: cardinality of the set M_c (bounded above by $|E_c|$)

The maximum number of operations required for each step is given below, within a constant factor.

Step 3 : $|X_c| * |C_1| * arity + |Y_c| * |C_2| * arity$

Step 4 : $|X_c| * |Y_c| * |C_1| * |C_2|$

Step 5 : $|E_c| * |C_1| * arity * |C_2| * arg$

Step 6 : $|X_a| * |Y_a| * |Y_a| + (|X_a| + |Y_a|) * (|X_c| + |E_c|)$

Step 7 : $|Y_a| * |X_a| * |X_a| + (|X_a| + |Y_a|) * (|Y_c| + |E_c|)$

Step 8 : $|E_a| * (|X_a| + |Y_a|) * \log_{(|E_a|/(|X_a|+|Y_a|)+1)}(|X_a| + |Y_a|)$

Step 9 : $|E_c| * |M_a| * |C_1| * |C_2| * arity * arg$

Step 10 : $|M_a| * |M_c| * (|C_1| + |C_2|) * arity * arg$

Step 11 : $|M_c|$

Step 12 : *constant*

Step 13 : $|M_c| * arg * (|C_1| + |C_2|) * arity$

Step 14 : Each of steps 4 through 7 of the unskolemization algorithm take time $|M_c| * (|C_1| + |C_2|) * arity$, for the generation of one unskolemized formula. •

Working of problem from Section 4.7 using the algorithm LEARN

We process the examples in the order E_1, E_2 and E_3 . The examples are shown in Figure 4.10. The same result is obtained for other orders of presentation of the examples.

First the examples E_1 and E_2 are taken and all possible resolutions are performed between these two examples and the given axioms. The resulting sets of clauses obtained from E_1 and E_2 , called X_c and Y_c respectively, are :

$$X_c = medium(a) \wedge polygon(a) \wedge blank(a) \wedge ontop(a, b) \wedge medium(b) \wedge circle(b) \wedge shaded(b) \wedge ontop(b, c) \wedge large(c) \wedge polygon(c) \wedge blank(c),$$

$$Y_c = medium(d) \wedge polygon(d) \wedge blank(d) \wedge ontop(d, e) \wedge small(f) \wedge circle(f) \wedge shaded(f) \wedge inside(f, e) \wedge small(g) \wedge circle(g) \wedge shaded(g) \wedge inside(g, e) \wedge large(e) \wedge polygon(e) \wedge blank(e).$$

We build the clause and argument graphs for X_c and Y_c ; these graphs are shown in Figures A.1 and A.2 respectively. These graphs do not need to be augmented since neither X_c nor Y_c contain any variables. There exists two maximum weight matchings for the argument graph; these are

$$\{(a, d)(4), (b, f)(2), (c, e)(4)\} \text{ and } \{(a, d)(4), (b, g)(2), (c, e)(4)\}$$

(the weights for each edge are indicated after each edge in parentheses). However, it turns out that both these matchings give rise to the same concept. We therefore choose the first matching and get

$$M_a = \{(a, d), (b, f), (c, e)\}.$$

The set M_c contains the edges which are shown in Figure A.3. We then replace a and d by $X \leftarrow a$ and $X \leftarrow d$ respectively; we replace b and f by $Y \leftarrow b$ and $Y \leftarrow f$ respectively; and we replace c and e by $Z \leftarrow c$ and $Z \leftarrow e$ respectively in the edges of M_c . We then get

$$EX = \{\{medium(X \leftarrow a), medium(X \leftarrow d)\}, \{polygon(X \leftarrow a), polygon(X \leftarrow d)\}, \{blank(X \leftarrow a), blank(X \leftarrow d)\}, \{ontop(X \leftarrow a, Y \leftarrow b), ontop(X \leftarrow d, Z \leftarrow e)\}, \{circle(Y \leftarrow b), circle(Y \leftarrow f)\}, \{shaded(Y \leftarrow b), shaded(Y \leftarrow f)\}, \{ontop(Y \leftarrow b, Z \leftarrow c), ontop(X \leftarrow d, Z \leftarrow e)\}, \{large(Z \leftarrow c), large(Z \leftarrow e)\}, \{blank(Z \leftarrow c), blank(Z \leftarrow e)\}, \{polygon(Z \leftarrow c), polygon(Z \leftarrow e)\}\}.$$

We now unskolemize EX by replacing the marked arguments by existentially quantified variables and get

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge circle(Y) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z) \wedge polygon(Z)).$$

This is the concept learned from E_1 and E_2 . We will now apply the algorithm to E_3 and the above formula EX . First we perform all possible resolutions between the axioms and these examples. EX remains unchanged; we get the following set of clauses from E_3 :

$$E_3 = \{ \{medium(h)\}, \{polygon(h)\}, \{blank(h)\}, \{ontop(h, j)\}, \{medium(j)\}, \\ \{polygon(j)\}, \{shaded(j)\}, \{ontop(j, k)\}, \{large(k)\}, \{ellipse(k)\}, \\ \{blank(k)\} \}.$$

We now need to express EX in clause form. After Skolemizing EX , we get the set of clauses

$$\{ \{medium(s)\}, \{polygon(s)\}, \{blank(s)\}, \{ontop(s, t), ontop(s, u)\}, \{circle(t)\}, \\ \{shaded(t)\}, \{ontop(t, u), ontop(s, u)\}, \{large(u)\}, \{blank(u)\} \},$$

where s, t, u are Skolem functions replacing the existentially quantified variables X, Y, Z respectively.

We build the clause and argument graphs for these two sets of clauses; these graphs are shown in Figures A.4 and A.5 respectively. These graphs do not need to be augmented since none of the clauses contain any variables. The maximum weight matching for the argument graph is :

$$M_a = \{(h, s)(5), (j, t)(3), (k, u)(5)\}$$

(the weights for each edge are indicated after each edge in parentheses). The set M_c contains the edges which are shown in Figure A.6. We then replace h and s by $X \leftarrow h$ and $X \leftarrow s$ respectively; we replace j and t by $Y \leftarrow j$ and $Y \leftarrow t$ respectively; and we replace k and u by $Z \leftarrow k$ and $Z \leftarrow u$ respectively in the edges of M_c . We then get

$$EX = \{ \{medium(X \leftarrow h), medium(X \leftarrow s)\}, \{polygon(X \leftarrow h), polygon(X \leftarrow s)\}, \\ \{blank(X \leftarrow h), blank(X \leftarrow s)\}, \{ontop(X \leftarrow h, Y \leftarrow j), ontop(X \leftarrow s, Y \leftarrow t), \\ ontop(X \leftarrow s, Z \leftarrow u)\}, \{ontop(X \leftarrow h, Y \leftarrow j), ontop(Y \leftarrow t, Z \leftarrow u), \\ ontop(X \leftarrow s, Z \leftarrow u)\}, \{shaded(Y \leftarrow j), shaded(Y \leftarrow t)\}, \{ontop(Y \leftarrow j, Z \leftarrow k), \\ ontop(Y \leftarrow t, Z \leftarrow u), ontop(X \leftarrow s, Z \leftarrow u)\}, \{ontop(Y \leftarrow j, Z \leftarrow k), ontop(X \leftarrow s, Y \leftarrow t), \\ ontop(X \leftarrow s, Z \leftarrow u)\}, \{large(Z \leftarrow k), large(Z \leftarrow u)\}, \{blank(Z \leftarrow k), blank(Z \leftarrow u)\} \}.$$

We now unskolemize EX by replacing the marked arguments by existentially quantified variables and get

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge (ontop(X, Y) \vee ontop(Y, Z) \vee ontop(X, Z)) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z)).$$

This is the concept learned from E_1 , E_2 and E_3 . Note that one of the disjunctions here is subsumed by two of the others, namely the disjunction $(ontop(X, Y) \vee ontop(Y, Z) \vee ontop(X, Z))$; therefore it can be discarded. The resulting concept learned from the three given examples is

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z)). \bullet$$

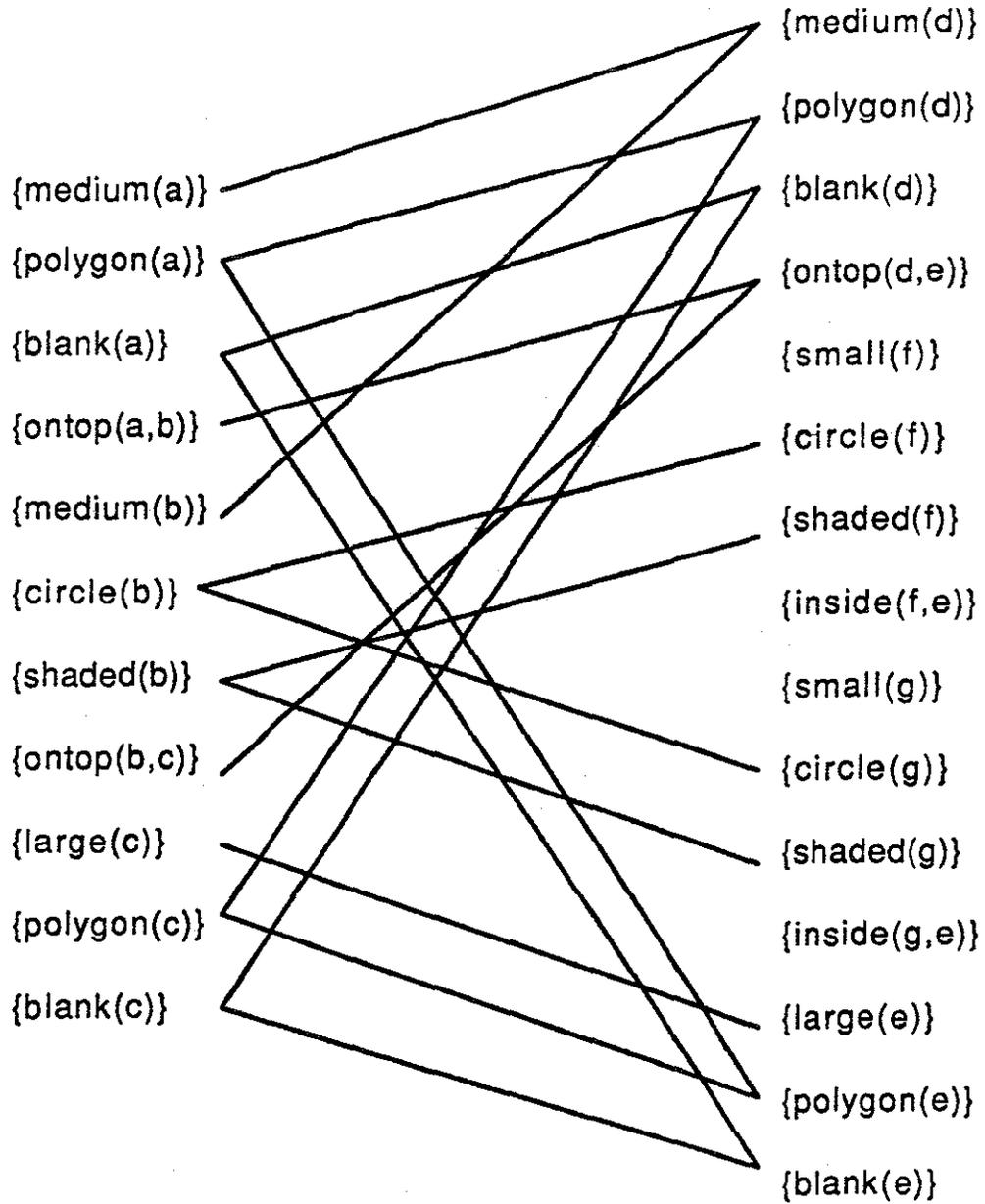


Figure A.1 Clause graph for E_1 and E_2

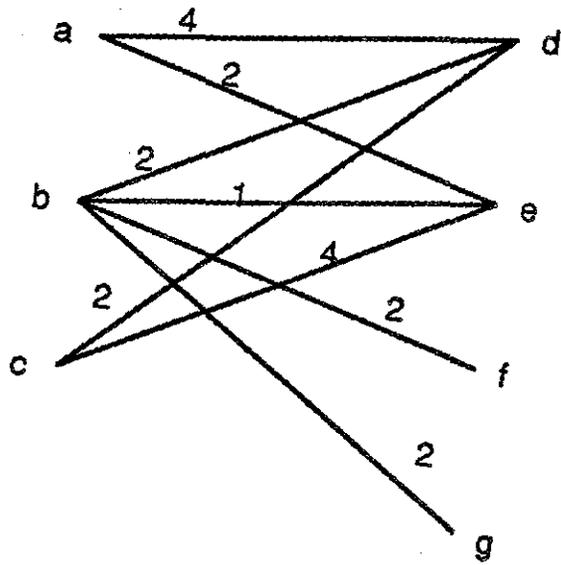


Figure A.2 Argument graph for E_1 and E_2

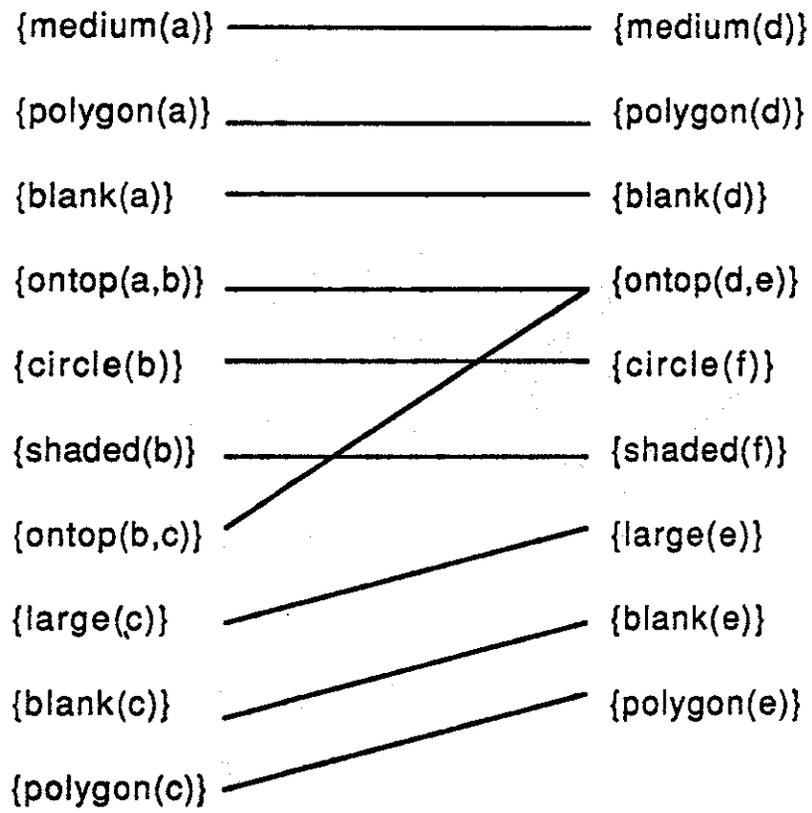


Figure A.3 Subgraph of clause graph for E_1 and E_2

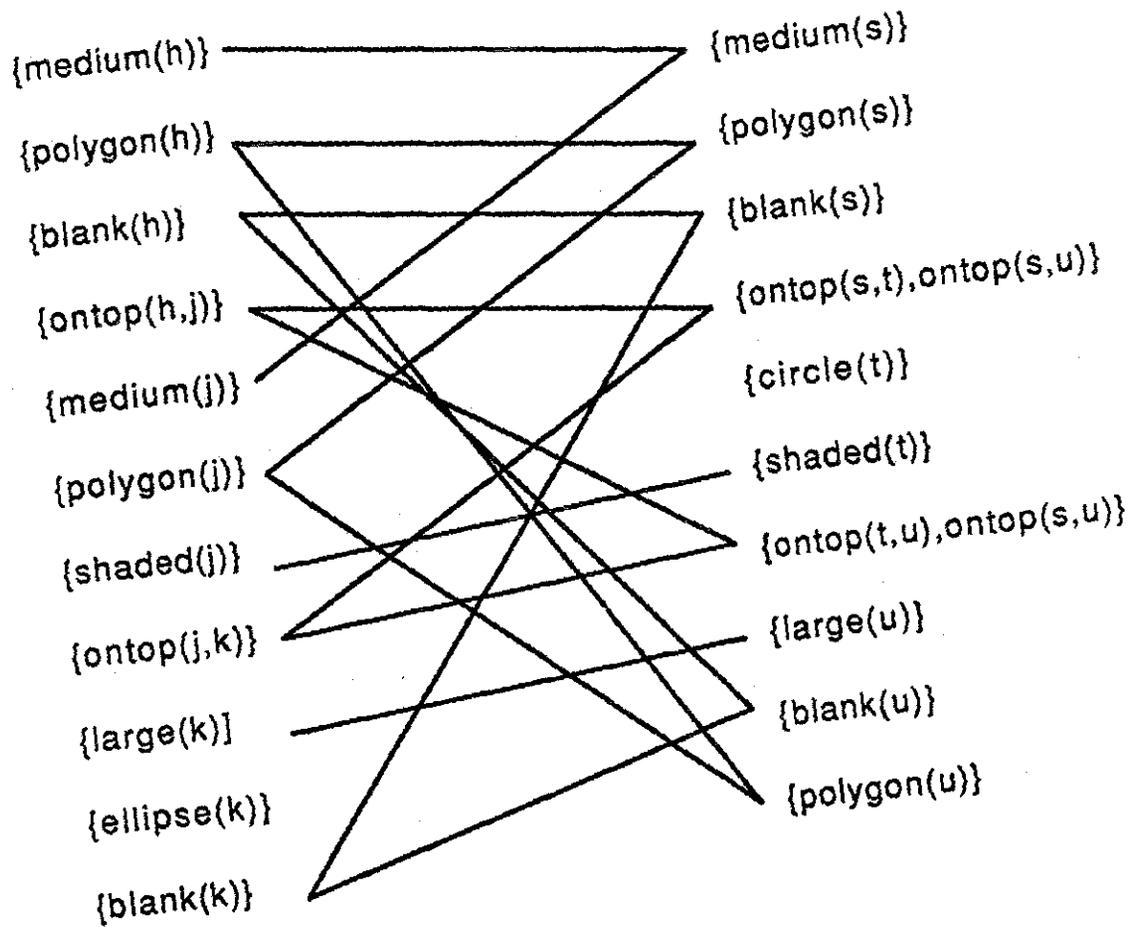


Figure A.4 Clause graph for E_3 and EX

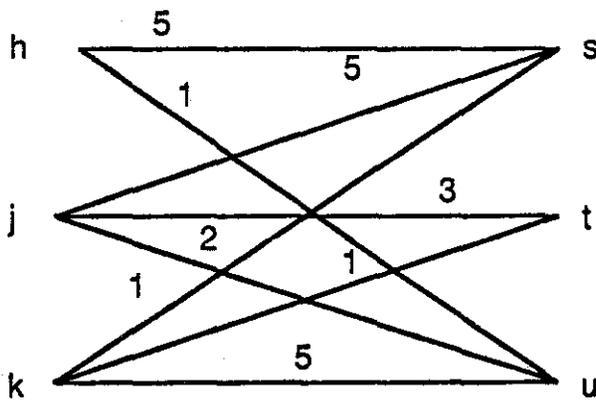


Figure A.5 Argument graph for E_3 and EX

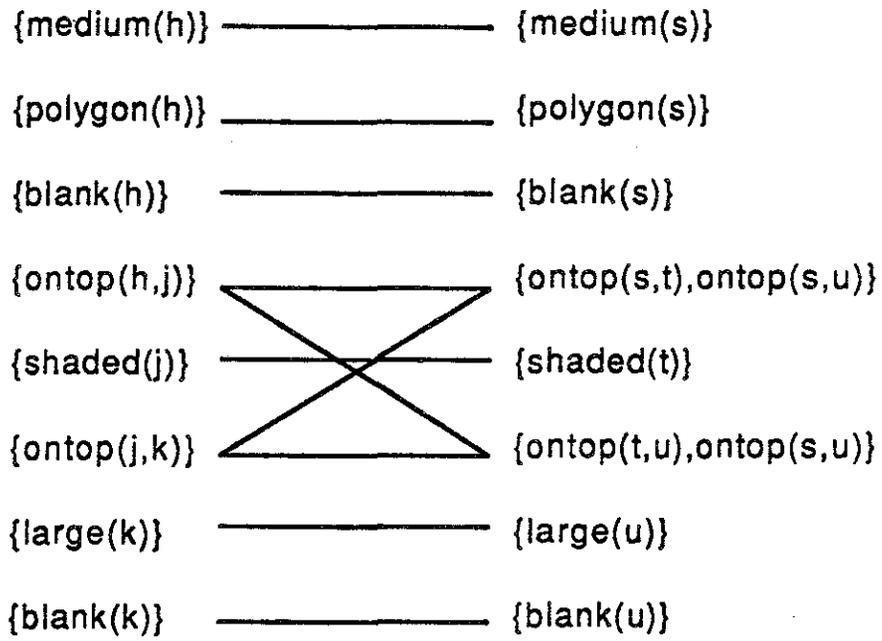


Figure A.6 Subgraph of clause graph for E_3 and EX

Proofs for Example 5.2

The following proofs were obtained for Example 5.2 using the theorem prover OTTER :

Proof of base step from Example 5.2 :

- 1 [] (X = X).
- 2 [] (append(nil,Y) = Y).
- 7 [] (reverse(append(nil,cons(y,nil))) != cons(y,reverse(nil))).
- 8 [] (reverse(nil) = nil).
- 11 [] (reverse(cons(X,nil)) = cons(X,nil)).
- 12 [para_into,7,2,demod,11,8] (cons(y,nil) != cons(y,nil)).
- 13 [binary,12,1] .

Proof of induction step from Example 5.2 :

- 1 [] (X = X).
- 3 [] (append(X,Y) = cons(car(X),append(cdr(X),Y))) | -listp(X).
- 4 [] (reverse(X) = append(reverse(cdr(X)),cons(car(X),nil))) | -listp(X).
- 5 [] listp(cons(X,Y)).
- 7 [] (reverse(append(cdr(x),cons(y,nil))) = cons(y,reverse(cdr(x)))).
- 8 [] listp(x).
- 9 [] (reverse(append(x,cons(y,nil))) != cons(y,reverse(x))).
- 11 [] (car(cons(X,Y)) = X).
- 12 [] (cdr(cons(X,Y)) = Y).
- 14 [para_into,9,3] (reverse(cons(car(x), append(cdr(x), cons(y, nil)))) != cons(y, reverse(x))) | -listp(x).
- 17 [binary,14,8] (reverse(cons(car(x), append(cdr(x),cons(y, nil)))) != cons(y, reverse(x))).
- 19 [para_into,17,4,demod,12,11] (append(reverse(append(cdr(x), cons(y,nil))), cons(car(x), nil)) != cons(y, reverse(x))) | -listp(cons(car(x), append(cdr(x), cons(y, nil))))).
- 54 [binary,19,5] (append(reverse(append(cdr(x), cons(y, nil))),cons(car(x), nil)) != cons(y, reverse(x))).
- 56 [para_into,54,7] (append(cons(y, reverse(cdr(x))), cons(car(x), nil)) != cons(y, reverse(x))).
- 61 [para_into,56,3,demod,11,12] (cons(y, append(reverse(cdr(x)), cons(car(x), nil))) != cons (y, reverse(x))) | -listp(cons(y, reverse(cdr(x)))).
- 63 [binary,61,5] (cons(y, append(reverse(cdr(x)), cons(car(x), nil))) != cons(y, reverse(x))).

65 [para.into,63,4] (cons(y,reverse(x)) != cons(y,reverse(x))) | -listp(x).
 68 [binary,65,1] -listp(x).
 69 [binary,68,8]. •

Working of examples for Section 5.4

We give below three examples illustrating the technique outlined in Section 5.4 for discovering inductive hypotheses.

Example 5.4 To illustrate the discussion in Section 5.4, suppose that we are trying to prove the commutativity of addition, using the Peano axioms :

1. $\forall x((x = 0) \vee (x = a + 1))$ where a is a Skolem symbol
2. $\forall x \forall y((x \neq y + 1) \vee (x \neq 0))$
3. $\forall x \forall y((x + 1 \neq y + 1) \vee (x = y))$
4. $\forall x \forall y(\neg(x < y + 1) \vee (x < y) \vee (x = y))$
5. $\forall x \forall y(\neg(x < y) \vee (x < y + 1))$
6. $\forall x \forall y((x \neq y) \vee (x < y + 1))$
7. $\forall x(\neg(x < 0))$
8. $\forall x \forall y((x < y) \vee (x = y) \vee (y < x))$
9. $\forall x(x + 0 = x)$
10. $\forall x \forall y(x + (y + 1) = (x + y) + 1)$
11. $\forall x(x * 0 = 0)$
12. $\forall x \forall y(x * (y + 1) = x * y + x)$
13. $\forall x(x = x)$

The theorem to be proved is

$$\forall x \forall y(x + y = y + x)$$

An attempt to prove this theorem without induction, using only the above axioms and resolution, fails. We therefore start trying to prove ground instances of the theorem. Three ground proofs are shown below :

1) **Proof of $(1+1) + (((1+1)+1)+1) = (((1+1)+1)+1) + (1+1)$.**

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{(1 + 1) + (((1 + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$

1. $\{(((1 + 1) + ((1 + 1) + 1)) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1))\}$

paramodulate with axiom 10

2. $\{(((1 + 1) + (1 + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$

- paramodulate with axiom 10
3. $\{((((1 + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 4. $\{(((1 + 1) + 1) + (1 + 1)) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 5. $\{((((1 + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 6. $\{(((1 + 1) + 1) + 1) + (1 + 1) \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 7. $\{\}$ resolve with axiom 13

2) Proof of $(1+1) + (((1+1)+1)+1)+1 = (((1+1)+1)+1)+1 + (1+1)$.

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{(1+1)+((((1+1)+1)+1)+1) \neq (((1+1)+1)+1)+1+(1+1)\}$

1. $\{(1 + 1) + (((1 + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
2. $\{((1 + 1) + (((1 + 1) + 1) + 1)) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
3. $\{(((1 + 1) + ((1 + 1) + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
4. $\{((((1 + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
5. $\{((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
6. $\{((((((1 + 1) + 1) + (1 + 1)) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
7. $\{(((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
8. $\{(((((((1 + 1) + 1) + 1) + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
9. $\{(((((((1 + 1) + 1) + 1) + 1) + 1) + (1 + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
10. $\{\}$ resolve with axiom 13.

3) Proof of $((1+1)+1) + (((1+1)+1)+1)+1 = (((1+1)+1)+1)+1 + ((1+1) + 1)$.

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{((1 + 1) + 1) + (((1 + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$

1. $\{(((1 + 1) + 1) + ((1 + 1) + 1)) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
2. $\{((((1 + 1) + 1) + ((1 + 1) + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
3. $\{((((((1 + 1) + 1) + (1 + 1)) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
4. $\{(((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
5. $\{((((((1 + 1) + 1) + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
6. $\{((((((1 + 1) + 1) + 1) + ((1 + 1) + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
7. $\{(((((((1 + 1) + 1) + 1) + (1 + 1)) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
8. $\{((((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
9. $\{((((((((1 + 1) + 1) + 1) + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
10. $\{((((((((1 + 1) + 1) + 1) + 1) + 1) + ((1 + 1) + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
11. $\{\}$ resolve with axiom 13.

It can be seen that each proof contains an instance of the proof of the lemma

$$\forall x \forall y ((y - 1) + x) + 1 = ((y - 1) + 1) + x,$$

namely in clauses 4, 8, and 6 for the three proofs respectively.

This lemma can easily be proved by induction, using the well-founded order $<$. The given theorem can then be proved by induction, using this lemma as an axiom. •

Example 5.5 Let us try to prove the theorem

$$\forall x \forall y (x * y = y * x).$$

An attempt to prove this theorem by first-order methods, using the Peano axioms given in Example 5.4, fails. We therefore try to prove ground instances of the theorem. We assume that the following simple theorem has already been

proved : $\forall x(0 + x = x)$. Some proofs of ground instances of the theorem are given below :

1) Proof of $1*(1+1) = (1+1)*1$.

Negation of the theorem : $1*(1+1) \neq (1+1)*1$.

The proof proceeds as follows :

1. $\{(1*1) + 1 \neq (1+1)*1\}$
paramodulate with axiom 12
2. $\{(1*(0+1)) + 1 \neq (1+1)*1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
3. $\{((1*0) + 1) + 1 \neq (1+1)*1\}$
paramodulate with axiom 12
4. $\{(0+1) + 1 \neq (1+1)*1\}$
paramodulate with axiom 11
5. $\{0 + (1+1) \neq (1+1)*1\}$
paramodulate with axiom 10
6. $\{((1+1)*0) + (1+1) \neq (1+1)*1\}$
paramodulate with axiom 11
7. $\{(1+1)*(0+1) \neq (1+1)*1\}$
paramodulate with axiom 12
8. $\{(1+1)*1 \neq (1+1)*1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
9. $\{\}$ paramodulate with axiom 13.

2) Proof of $1*((1+1)+1) = ((1+1)+1)*1$.

Negation of the theorem : $1*((1+1)+1) \neq ((1+1)+1)*1$.

The proof proceeds as follows :

1. $\{(1*(1+1)) + 1 \neq ((1+1)+1)*1\}$
paramodulate with axiom 12
2. $\{((1*1) + 1) + 1 \neq ((1+1)+1)*1\}$
paramodulate with axiom 12
3. $\{((1*(0+1)) + 1) + 1 \neq ((1+1)+1)*1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
4. $\{(((1*0) + 1) + 1) + 1 \neq ((1+1)+1)*1\}$
paramodulate with axiom 12
5. $\{((0+1) + 1) + 1 \neq ((1+1)+1)*1\}$
paramodulate with axiom 11
6. $\{(0 + (1+1)) + 1 \neq ((1+1)+1)*1\}$
paramodulate with axiom 10
7. $\{(((1+1)*0) + (1+1)) + 1 \neq ((1+1)+1)*1\}$

- paramodulate with axiom 11
8. $\{((1 + 1) * (0 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
9. $\{((1 + 1) * 1) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
10. $\{((1 + 1) * (0 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
11. $\{(((1 + 1) * 0) + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
12. $\{(0 + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 11
13. $\{0 + ((1 + 1) + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 10
14. $\{(((1 + 1) + 1) * 0) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 11
15. $\{((1 + 1) + 1) * (0 + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
16. $\{((1 + 1) + 1) * 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
17. $\{\}$ resolve with axiom 13

3) Proof of $(1+1)*((1+1)+1) = ((1+1)+1)*(1+1)$.

Negation of the theorem : $(1 + 1) * ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)$.

The proof proceeds as follows :

1. $\{((1 + 1) * (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
2. $\{(((1 + 1) * 1) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
3. $\{(((1 + 1) * (0 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
4. $\{((((1 + 1) * 0) + (1 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
5. $\{(((0 + (1 + 1)) + (1 + 1)) + (1 + 1)) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 11
6. $\{(((1 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
7. $\{(((1 + 1) + 1) + 1) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 10
8. $\{(((1 + 1) + 1) + (1 + 1)) + 1 \neq ((1 + 1) + 1) * (1 + 1)\}$

- paramodulate with axiom 10
9. $\{((1 + 1) + 1) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 10
10. $\{(0 + ((1 + 1) + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
11. $\{((((1 + 1) + 1) * 0) + ((1 + 1) + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 11
12. $\{(((1 + 1) + 1) * (0 + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
13. $\{(((1 + 1) + 1) * 1) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
14. $\{((1 + 1) + 1) * (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
15. $\{\}$ resolve with axiom 13.

From the proofs of the three above ground instances, we see that each proof contains a subproof of the lemma

$$\forall x \forall y ((x + 1) * y = (x * y) + y),$$

namely in clauses 1, 9, and 1 respectively of the three proofs.

And this lemma can be proved by induction, using the well-founded order $<$. The given theorem can then be proved by induction, using this lemma as an axiom. •

Example 5.6 In this example, we solve the same problem as that solved in Example 5.3 in Chapter 5, i.e. we are trying to prove the theorem

$$\forall x(\text{reverse}(\text{reverse}(x)) = x)$$

by induction, this time using the method described in this section. A well-founded ordering for this example was already discovered in Example 5.1 in Chapter 5.

We prove the theorem for different values of ground x :

Proof of $\text{reverse}(\text{reverse}([a, b])) = [a, b]$:

The axioms used in this proof are (in clause form) :

1. $X = X$
2. $\text{append}(\text{nil}, Y) = Y$
3. $\text{append}(X, Y) = \text{cons}(\text{car}(X), \text{append}(\text{cdr}(X), Y)) \vee \neg \text{listp}(X)$
4. $\text{reverse}(\text{nil}) = \text{nil}$
5. $\text{reverse}(X) = \text{append}(\text{reverse}(\text{cdr}(X)), \text{cons}(\text{car}(X), \text{nil})) \vee \neg \text{listp}(X)$
6. $\text{car}(\text{cons}(X, Y)) = X$
7. $\text{cdr}(\text{cons}(X, Y)) = Y$

8. $listp(cons(X, Y))$

Negation of theorem :

$$reverse(reverse([a, b])) \neq [a, b]$$

Using a set of support strategy, we get the following proof for the theorem :

9. $reverse(reverse([a, b])) \neq [a, b]$

negation of theorem

10. $reverse(append(reverse(cdr([a, b])), cons(car([a, b]), nil))) \neq [a, b],$

$\neg listp([a, b])$

paramodulate 9,5

11. $reverse(append(reverse(cdr([a, b])), cons(car([a, b]), nil))) \neq [a, b]$

resolve 8,10

12. $reverse(append(reverse([b]), cons(car([a, b]), nil))) \neq [a, b]$

paramodulate 7,11

13. $reverse(append(reverse([b]), [a])) \neq [a, b]$

paramodulate 6,12

14. $reverse(append(append(reverse(cdr([b])), cons(car([b]), nil)), [a])) \neq [a, b],$

$\neg listp([b])$

paramodulate 5,13

15. $reverse(append(append(reverse(cdr([b])), [b]), [a])) \neq [a, b], \neg listp([b])$

paramodulate 6,14

16. $reverse(append(append(reverse(nil), [b]), [a])) \neq [a, b], \neg listp([b])$

paramodulate 7,15

17. $reverse(append(append(reverse(nil), [b]), [a])) \neq [a, b]$

resolve 8,16

18. $reverse(append(append(nil, [b]), [a])) \neq [a, b]$

paramodulate 4,17

19. $reverse(append([b], [a])) \neq [a, b]$

paramodulate 2,18

20. $reverse(cons(car([b]), append(cdr([b]), [a]))) \neq [a, b], \neg listp([b])$

paramodulate 3,19

21. $reverse(cons(car([b]), append(nil, [a]))) \neq [a, b], \neg listp([b])$

paramodulate 7,20

22. $reverse(cons(car([b]), append(nil, [a]))) \neq [a, b]$

resolve 8,21

23. $reverse(cons(car([b]), [a])) \neq [a, b]$

paramodulate 2,22

24. $reverse([b, a]) \neq [a, b]$

- paramodulate 6,23
25. $append(reverse(cdr([b, a])), cons(car(cons(b, [a]))) \neq [a, b], \neg listp([b])$
 paramodulate 5,24
26. $append(reverse([a]), cons(car([b, a]), nil)) \neq [a, b], \neg listp([b])$
 paramodulate 7,25
27. $append(reverse([a]), [b]) \neq [a, b], \neg listp([b])$
 paramodulate 6,26
28. $append(reverse([a]), [b]) \neq [a, b]$
 resolve 8,27
29. $append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]) \neq [a, b], \neg listp([a])$
 paramodulate 5,28
30. $append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]) \neq [a, b]$
 resolve 8,29
31. $append(append(reverse(nil), cons(car([a]), nil)), [b]) \neq [a, b]$
 paramodulate 7,30
32. $append(append(reverse(nil), [a]), [b]) \neq [a, b]$
 paramodulate 6,31
33. $append(append(nil, [a]), [b]) \neq [a, b]$
 paramodulate 4,32
34. $append([a], [b]) \neq [a, b]$
 paramodulate 2,33
35. $cons(car([a]), append(cdr([a]), [b])) \neq [a, b], \neg listp([a])$
 paramodulate 3,34
36. $cons(car([a]), append(cdr([a]), [b])) \neq [a, b]$
 resolve 8,35
37. $cons(a, append(cdr([a]), [b])) \neq [a, b]$
 paramodulate 6,36
38. $cons(a, append(nil, [b])) \neq [a, b]$
 paramodulate 7,37
39. $[a, b] \neq [a, b]$
 paramodulate 2,38
40. empty clause
 resolve 1,39.

Proof of $reverse(reverse([a, b, c])) = [a, b, c]$:

Using a set of support strategy, and the same axioms (1 through 8 above), we get the following proof for the theorem :

9. $reverse(reverse([a, b, c])) \neq [a, b, c]$

negation of theorem

10. $reverse(append(reverse(cdr([a, b, c])), cons(car([a, b, c]), nil))) \neq [a, b, c],$
 $\neg listp([a, b, c])$
paramodulate 5,9
11. $reverse(append(reverse([b, c]), cons(car([a, b, c]), nil))) \neq [a, b, c],$
 $\neg listp([a, b, c])$
paramodulate 7,10
12. $reverse(append(reverse([b, c]), [a])) \neq [a, b, c], \neg listp([a, b, c])$
paramodulate 6,11
13. $reverse(append(reverse([b, c]), [a])) \neq [a, b, c]$
resolve 8,12
14. $reverse(append(append(reverse(cdr([b, c])), cons(car([b, c]), nil)), [a])) \neq$
 $[a, b, c], \neg listp([b, c])$
paramodulate 5,13
15. $reverse(append(append(append(reverse(cdr([b, c])), cons(car([b, c]), nil)), [a])) \neq$
 $[a, b, c]$
resolve 8,14
16. $reverse(append(append(reverse([c]), cons(car([b, c]), nil)), [a])) \neq [a, b, c]$
paramodulate 7,15
17. $reverse(append(append(reverse([c]), [b]), [a])) \neq [a, b, c]$
paramodulate 6,16
18. $reverse(append(append(append(reverse(cdr([c])), cons(car([c]), nil)), [b]), [a]))$
 $\neq [a, b, c], \neg listp([c])$
paramodulate 5,17
19. $reverse(append(append(append(reverse(cdr([c])), [c]), [b]), [a])) \neq [a, b, c],$
 $\neg listp([c])$
paramodulate 6,18
20. $reverse(append(append(append(reverse(nil), [c]), [b]), [a])) \neq [a, b, c],$
 $\neg listp([c])$
paramodulate 7,19
21. $reverse(append(append(append(reverse(nil), [c]), [b]), [a])) \neq [a, b, c]$
resolve 8,20
22. $reverse(append(append(append(nil, [c]), [b]), [a])) \neq [a, b, c]$
paramodulate 4,21
23. $reverse(append(append([c], [b]), [a])) \neq [a, b, c]$
paramodulate 2,22
24. $reverse(append(cons(car([c]), append(cdr([c]), [b])), [a])) \neq [a, b, c], \neg listp([c])$
paramodulate 3,23

25. $reverse(append(cons(car([c]), append(cdr([c]), [b])), [a])) \neq [a, b, c]$
 resolve 8,24
26. $reverse(append(cons(c, append(cdr([c]), [b])), [a])) \neq [a, b, c]$
 paramodulate 6,25
27. $reverse(append(cons(c, append(nil, [b])), [a])) \neq [a, b, c]$
 paramodulate 7,26
28. $reverse(append([c, b], [a])) \neq [a, b, c]$
 paramodulate 2,27
29. $reverse(cons(car([c, b]), append(cdr([c, b]), [a]))) \neq [a, b, c], \neg listp([c, b])$
 paramodulate 3,28
30. $reverse(cons(car([c, b]), append(cdr([c, b]), [a]))) \neq [a, b, c]$
 resolve 8,29
31. $reverse(cons(c, append(cdr([c, b]), [a]))) \neq [a, b, c]$
 paramodulate 6,30
32. $reverse(cons(c, append([b], [a]))) \neq [a, b, c]$
 paramodulate 7,31
33. $reverse(cons(c, cons(car([b]), append(cdr([b]), [a]))) \neq [a, b, c], \neg listp([b])$
 paramodulate 3,32
34. $reverse(cons(c, cons(car([b]), append(cdr([b]), [a]))) \neq [a, b, c]$
 resolve 8,33
35. $reverse(cons(c, cons(b, append(cdr([b]), [a]))) \neq [a, b, c]$
 paramodulate 6,34
36. $reverse(cons(c, cons(b, append(nil, [a]))) \neq [a, b, c]$
 paramodulate 7,35
37. $reverse([c, b, a]) \neq [a, b, c]$
 paramodulate 2,36
38. $append(reverse(cdr([c, b, a])), cons(car([c, b, a]), nil)) \neq [a, b, c], \neg listp([c, b, a])$
 paramodulate 5,37
39. $append(reverse(cdr([c, b, a])), cons(car([c, b, a]), nil)) \neq [a, b, c]$
 resolve 8,38
40. $append(reverse([b, a]), cons(car([c, b, a]), nil)) \neq [a, b, c]$
 paramodulate 7,39
41. $append(reverse([b, a]), [c]) \neq [a, b, c]$
 paramodulate 6,40
42. $append(append(reverse(cdr([b, a])), cons(car([b, a]), nil)), [c]) \neq [a, b, c],$
 $\neg listp([b, a])$
 paramodulate 5,41
43. $append(append(reverse([a]), cons(car([b, a]), nil)), [c]) \neq [a, b, c], \neg listp([b, a])$

- paramodulate 7,42
44. $append(append(reverse([a]), [b]), [c]) \neq [a, b, c], \neg listp([b, a])$
paramodulate 6,43
45. $append(append(reverse([a]), [b]), [c]) \neq [a, b, c]$
resolve 8,44
46. $append(append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]), [c])$
 $\neq [a, b, c], \neg listp([a])$
paramodulate 5,45
47. $append(append(append(reverse(nil), cons(car([a]), nil)), [b]), [c]) \neq [a, b, c],$
 $\neg listp([a])$
paramodulate 7,46
48. $append(append(append(reverse(nil), [a]), [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 6,47
49. $append(append(append(nil, [a]), [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 4,48
50. $append(append([a], [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 2,49
51. $append(append([a], [b]), [c]) \neq [a, b, c]$
resolve 8,50
52. $append(cons(car([a]), append(cdr([a]), [b])), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 3,51
53. $append(cons(car([a]), append(cdr([a]), [b])), [c]) \neq [a, b, c]$
resolve 8,52
54. $append(cons(a, append(cdr([a]), [b])), [c]) \neq [a, b, c]$
paramodulate 6,53
55. $append(cons(a, append(nil, [b])), [c]) \neq [a, b, c]$
paramodulate 7,54
56. $append([a, b], [c]) \neq [a, b, c]$
paramodulate 2,55
57. $cons(car([a, b]), append(cdr([a, b]), [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 3,56
58. $cons(a, append(cdr([a, b]), [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 6,57
59. $cons(a, append([b], [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 7,58
60. $cons(a, append([b], [c])) \neq [a, b, c]$
resolve 8,59
61. $cons(a, cons(car([b]), append(cdr([b]), [c]))) \neq [a, b, c], \neg listp([b])$

- paramodulate 3,60
62. $\text{cons}(a, \text{cons}(b, \text{append}(\text{cdr}([b]), [c]))) \neq [a, b, c], \neg \text{listp}([b])$
 paramodulate 6,61
63. $\text{cons}(a, \text{cons}(b, \text{append}(\text{nil}, [c]))) \neq [a, b, c], \neg \text{listp}([b])$
 paramodulate 7,62
64. $\text{cons}(a, \text{cons}(b, \text{append}(\text{nil}, [c]))) \neq [a, b, c]$
 resolve 8,63
65. $[a, b, c] \neq [a, b, c]$
 paramodulate 2,64
66. empty clause
 resolve 1,65.

From the above two ground proofs, it can be seen that instances of the lemma $\forall x \forall y (\text{reverse}(\text{append}(x, \text{cons}(y, \text{nil}))) = \text{cons}(y, \text{reverse}(x)))$ were proved in both proofs. This is a lemma which needs to be proved by induction and was proved in Example 5.2 in Chapter 5 earlier. The given theorem can then be proved by induction, using this lemma as an axiom. •

References

- Angluin, Dana, Smith, Carl H. : "Inductive Inference : Theory and Methods", *Computing Surveys* 15 (3), pp. 237-269 (1983).
- Aubin, Raymond : "Mechanizing Structural Induction Part I : Formal System", *Theoretical Computer Science* 9, pp. 329-345 (1979).
- Aubin, Raymond : "Mechanizing Structural Induction Part II : Strategies", *Theoretical Computer Science* 9, pp. 347-362 (1979).
- Biundo, S., Hummel, B., Hutter, D., Walther, C. : "The Karlsruhe Induction Theorem Proving System", *Eighth International Conference on Automated Deduction*, pp. 672-674 (1986).
- Bondy, J. A., Murty, U. S. R. : "Graph Theory with Applications", *Elsevier Science Publishing Co., Inc., New York* (1976).
- Boyer, Robert S., Moore, J. Strother : "A Computational Logic", *Academic Press, Inc., New York* (1979).
- Buchanan, B. G., Mitchell, Tom M. : "Model-directed learning of production rules", *Proceedings of the Workshop on Pattern-directed Inference Systems, Honolulu, Hawaii* (1977).
- Burstall, R. M. : "Proving properties of programs by structural induction", *Computer Journal* 12 (1), pp. 41-48 (1969).
- Caplain, Michel, "Finding invariant assertions for proving programs", *Proceedings of the International Conference on Reliable Software*, pp. 165-171 (1975).
- Chang, Chin-Lian, Lee, Richard Char-Tung, "Symbolic Logic and Mechanical Theorem Proving", *Academic Press Inc., New York* (1973).
- Cook, Stephen A., "Soundness and completeness of an axiom system for program verification", *SIAM Journal on Computing* 7 (1), pp. 70-90 (1978).
- Cooper, D. C., "Programs for Mechanical Program Verification", *Machine Intelligence* 6, pp. 43-59 (1971).
- Cox, P. T., Pietrzykowski, T., "A complete, nonredundant algorithm for reversed skolemization", *Theoretical Computer Science* 28, pp. 239-261 (1984).
- Davis, M., Putnam, H., Robinson, J., "The decision problem for exponential Diophantine equations", *Annals of Mathematics* 74, pp. 425-436 (1961).
- Deutsch, Laurence P., "An Interactive Program Verifier", *Ph.D. dissertation, University of California at Berkeley* (1973).

- Dietterich, Thomas G., Michalski, Ryszard S : "A Comparative Review of Selected Methods for Learning from Examples", in : *Machine Learning : An Artificial Intelligence Approach*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 41-82 (1983).
- Dijkstra, E. W., "Invariance and non-determinacy", *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, pp. 157-165 (1985).
- Dijkstra, E. W., "On the cruelty of really teaching computing science", *Communications of the ACM* **32** (12), pp. 1398-1404 (1989).
- Elsplas, Bernard, Levitt, Karl N., Waldinger, Richard J., Waksman, Abraham, "An Assessment of Techniques for Proving Program Correctness", *ACM Computing Surveys* **4** (2), pp. 97-147 (1972).
- Floyd, R. W., "Assigning meanings to programs", *Proceedings of the Symposium on Applied Mathematics*, American Mathematical Society **19**, pp. 19-32 (1967).
- Galil, Zvi, "Efficient Algorithms for Finding Maximum Matching in Graphs", *ACM Computing Surveys* **18** (1), pp. 23-38 (1986).
- German, Steven M., Wegbreit, Ben, "A Synthesizer of Inductive Assertions", *IEEE Transactions on Software Engg.*, Vol. SE-1 (1), pp. 68-75 (1975).
- Goguen, J. A. : "How to prove algebraic inductive hypotheses without induction", *Fifth International Conference on Automated Deduction*, pp. 356-373 (1980).
- Gold, E. Mark : "Language Identification in the Limit", *Information and Control* **10**, pp. 447-474 (1967).
- Good, Donald I., London, Ralph L., Bledsoe, W. W., "An Interactive Program Verification System", *IEEE Transactions on Software Engg.*, Vol. SE-1 (1) (1975).
- Good, Donald I., Cohen, R. M., Hoch, C. G., Hunter, L. W., Hare, D. F., "Report on the language Gypsy, Version 2.0", *Technical Report ICSCA-CMP-10, Certifiable Minicomputer, Project, ICSCA, The University of Texas at Austin* (1978).
- Good, Donald I., "Mechanical proofs about computer programs", *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, pp. 55-75 (1985).
- Gries, David, "The Science of Programming", Springer-Verlag (1981).
- Hayes-Roth, Frederick, McDermott, John : "An Interference Matching Technique for Inducing Abstractions", *Communications of the ACM* **21** (5), pp. 401-410 (1978).
- Hoare, C.A.R., "An axiomatic basis for computer programming", *Communications of the ACM* **12**, pp. 576-580 (1969).
- von Henke, F. W., Luckham, D. C., "A methodology for verifying programs", *Proceedings of the International Conference on Reliable Software*, pp. 156-164 (1975).

- Huet, G, Hullot, J.M. : "Proofs by induction in equational theories with constructors", *Journal of Computer and System Sciences* 25 (2) (1982).
- Jouannaud, Jean-Pierre, Kounalis, Emmanuel : "Automatic Proofs by Induction in Equational Theories Without Constructors", *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pp. 358-366 (1986).
- Kapur, Deepak, Musser, David R. : "Inductive reasoning with incomplete specifications" *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pp. 367-377 (1986).
- Katz, Shmuel M., Manna, Zohar, "A heuristic approach to program verification", *Third International Joint Conference on Artificial Intelligence*, pp. 500-512 (1973).
- King, James C., "A Program Verifier", *Ph.D. dissertation, Carnegie-Mellon University* (1969).
- King, James C., "An Interpretation Oriented Theorem Prover over Integers", *Second Annual ACM Symposium on Theory of Computing*, pp. 169-179 (1970).
- King, James C., "Proving Programs to be Correct", *IEEE Transactions on Computers*, Vol. C-20 (11), pp. 1331-1336 (1970).
- Knuth, D., Bendix, P. : "Simple Word Problems in Universal Algebras", in *Computational Problems in Abstract Algebra*, ed. J. Leech, Pergamon Press, pp. 263-297 (1970).
- Kodratoff, Y., Ganascia, J. G., Clavieras, B., Bollinger, T., Tecuci, G. : "Careful generalization for concept learning", in *Advances in Artificial Intelligence*, T. O'Shea (Ed.), Elsevier Science Publishers B.V. (North-Holland), pp. 229-238 (1985).
- Kodratoff, Y., Ganascia, J. G. : "Improving the Generalization Step in Learning", in : *Machine Learning : An Artificial Intelligence Approach*, Vol. II, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 215-244 (1986).
- Lee, Shie-Jue : "CLIN : An Automated Reasoning System Using Clause Linking", *Ph.D. Dissertation, University of North Carolina, Chapel Hill* (1990).
- Lewis, Harry R., Papadimitriou, Christos H., "Elements of the theory of computation", *New Jersey : Prentice-Hall, Inc.* (1981).
- Loeckx, Jacques, Sieber, Kurt, "The Foundations of Program Verification", *John Wiley and Sons, Ltd.* (1987).
- Loveland, Donald, "Automated Theorem Proving, A Logical Basis", *North-Holland Publishing Co.* (1978).
- Manna, Zohar, "Second-order Mathematical Theory of Computation", *Second Annual ACM Symposium on Theory of Computing*, pp. 158-168 (1970).
- Manna, Zohar, Ness, Stephen, Vuillemin, Jean : "Inductive Methods for Proving Properties of Programs", *Communications of the ACM* 16 (8), pp. 491-502 (1973).

- Manna, Zohar, "Mathematical Theory of Computation", McGraw-Hill, New York (1974).
- McCarthy, John : "A basis for a mathematical theory of computation", in *Computer Programming and Formal Systems*, eds. P. Braffort, D. Hirschberg, North-Holland Publishing Co., pp. 33-70 (1970).
- McCune, William W., "Un-Skolemizing clause sets", *Information Processing Letters* **29**, pp. 257-263 (1988).
- McCune, William W., "OTTER 1.0 Users' Guide", *Computer Science Division, Argonne National Laboratory, Argonne, Illinois* (1989).
- Michalski, Ryszard S. : "Learning by inductive inference", *Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-74-671* (1974).
- Michalski, Ryszard S. : "Toward Computer-Aided Induction : A brief review of currently implemented AQVAL programs", *Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-77-874* (1977).
- Michalski, Ryszard S. : "A Theory and Methodology of Inductive Learning", *Machine Learning : An Artificial Intelligence Approach, Vol. I*, eds. R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 83-134 (1983).
- Michalski, Ryszard S. : "Machine Learning : An Artificial Intelligence Approach", *Vol. II*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc. (1986).
- De Millo, R. A., Lipton, R. J., Perlis, A. J., "Social processes and proofs of theorems and programs", *Communications of the ACM* **22**, pp. 271-280 (1979).
- Mitchell, Tom M. : "Version Spaces : A candidate elimination approach to rule learning", *Fifth International Joint Conference on Artificial Intelligence, MIT, Cambridge MA, Vol. 1*, pp. 305-310 (1977).
- Mitchell, Tom M. : "Learning and Problem Solving", *Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, Germany*, pp. 1139-1151 (1983).
- Musser, David R. : "On proving inductive properties of abstract data types", *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 154-162 (1980).
- Nakajima, R., Yuasa, T., "The IOTA programming system", *Lecture Notes in Computer Science* **160**, Springer-Verlag (1983).
- Nelson, Greg, Oppen, Derek C., "Simplification by Cooperating Decision Procedures", *ACM Transactions on Programming Languages and Systems* **1** (2), pp. 245-257 (1979).
- Patterson, Dan W. : "Introduction to Artificial Intelligence and Expert Systems", Prentice Hall, Englewood Cliffs, New Jersey (1990).

- Paul, E. : "Proof by induction in equational theories with relations between constructors" *Ninth Colloquium on Trees in Algebra and Programming, Bordeaux, France*, ed. B. Courcelle, pp. 211-225 (1984).
- Polak, Wolfgang, "Compiler Specification and Verification", *Lecture Notes in Computer Science 124*, Springer - Verlag (1981).
- Robinson, J. A., "A Machine-oriented Logic based on the Resolution Principle", *Journal of the ACM 12 (1)*, pp. 23-41 (1965).
- Rulifson, J. F., Waldinger, R. J., Derksen, J., "A language for writing problem-solving programs", *IFIP Cong. 1971, Yugoslavia, North-Holland Publ. Co., Amsterdam (1972)*.
- Sammut, Claude, Banerji, Ranan B. : "Learning Concepts by Asking Questions", in : *Machine Learning : An Artificial Intelligence Approach, Vol. II*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 167-192 (1986).
- Sarkar, D., De Sarkar, S. C., "Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers", *IEEE Transactions on Software Engineering 15 (1)*, pp. 1-9 (1989).
- Sarkar, D., De Sarkar, S. C., "A Set of Inference Rules for Quantified Formula Handling and Array Handling in Verification of Programs over Integers", *IEEE Transactions on Software Engineering 15 (11)*, pp. 1368-1381 (1989).
- Sarkar, D., De Sarkar, S. C., "A Theorem Prover for Verifying Iterative Programs Over Integers", *IEEE Transactions on Software Engineering 15 (12)*, pp. 1550-1566 (1989).
- Shapiro, Ehud Y. : "An Algorithm that Infers Theories from Facts", *Seventh International Joint Conference on Artificial Intelligence*, pp. 446-451 (1981).
- Spitzen, Jay, Wegbreit, Ben, "The Verification and Synthesis of Data Structures", *Acta Informatica 4*, pp. 127-144 (1975).
- Stanford Verification Group, "Stanford Pascal Verifier User Manual", *Stanford Verification Group Report No. 11 (1979)*.
- Stepp, R. : "The investigation of the UNICLASS inductive program AQ7UNI and User's Guide", *Technical Report 949, Department of Computer Science, University of Illinois, Urbana, Illinois (1978)*.
- Suzuki, Narihisa, "Verifying programs by Algebraic and Logical Reduction", *Proceedings of the International Conference on Reliable Software*, pp. 473-481 (1975).
- Tinkham, Nancy : "Induction of Schemata for Program Synthesis", *Ph.D. dissertation, Department of Computer Science, Duke University, Durham N.C. (1990)*.
- Toyama, Y. : "How to prove equivalence of term rewriting systems without induction", *Eighth International Conference on Automated Deduction*, pp. 118-127 (1986).

- Utgoff, Paul E. : "Machine Learning of Inductive Bias", *Kluwer Academic Publishers, Massachusetts* (1986).
- Valiant, L. G. : "A Theory of the Learnable", *Communications of the ACM* 27 (11), pp. 1134-1142 (1984).
- Vanlehn, Kurt : "Efficient Specialization of Relational Concepts", *Machine Learning* 4, pp. 99-106 (1989).
- Vere, Steven A. : "Induction of Concepts in the Predicate Calculus", *Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia USSR, Vol. 1*, pp. 281-287 (1975).
- Vere, Steven A. : "Inductive learning of relational productions", in : *Pattern-Directed Inference Systems*, eds. D.A. Waterman, Frederick Hayes-Roth, Academic Press, Inc., New York, pp. 281-295 (1978).
- Wand, M., "A new incompleteness result for Hoare's system", *Journal of the ACM* 25, pp. 168-175 (1978).
- Watanabe, Larry, Rendell, Larry, "Effective Generalization of Relational Descriptions", *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 875-881 (1990).
- Wegbreit, Ben, "Heuristic Methods for Mechanically Deriving Inductive Assertions", *Proceedings of Third International Joint Conference on Artificial Intelligence* (1973).
- Wegbreit, Ben, Spitzen, Jay M. : "Proving Properties of Complex Data Structures", *Journal of the ACM* 23 (2), pp. 389-396 (1976).
- Winston, P. H. : "Learning structural descriptions from examples", in : *The Psychology of Computer Vision*, ed. P. H. Winston, McGraw-Hill, New York, pp. 157-209 (1975).
- Zhang, Hantao, Kapur, Deepak, Krishnamoorthy, Mukkai S. : "A mechanizable induction principle for equational specifications", *Ninth International Conference on Automated Deduction, Argonne, Illinois*, pp. 162-181 (1988).

then deduce that it holds for all data [Burstall 69]. This is a special case of a more general rule termed Noetherian induction : “Let A be an ordered set with minimum condition and B a subset of A which contains any element $a \in A$ whenever it contains all the elements $x \in A$ such that $x < a$. Then $B = A$.” (A set A satisfies the minimum condition if every non-empty subset of A has a minimal element.) The structural induction principle differs from the usual “course of values” induction in two ways : it allows for partial ordering, instead of total ordering, and it allows induction over the transfinite steps. Examples of some proofs by structural induction can be found in [Burstall 69].

In [Wegbreit and Spitzen 76], the authors introduce a method of proof called generator induction, used for proving properties of programs, which may be stated informally as follows : to prove that all instances of a class C have some property P , prove that (1) all instances have the property when they are first created, and (2) all operations F , which may change the value of a class instance, preserve the truth of P . This definition is similar to that of computation induction given in [Manna et al. 73]. The generator induction principle is used to prove certain properties of a hashtable program in Simula. The most important property of generator induction, according to the authors, is that it partitions the program into loosely coupled parts, proves simple properties of the parts, and demonstrates that the parts are composed according to simple rules. This allows the decomposition of a proof into small, comprehensible units corresponding to the structure of the program.

5.3 Description of the first method

We give below a brief overview of the first approach we will be using for deriving proofs of theorems. Consider what is provable by induction, where all induction hypotheses are expressible in first-order logic and all orderings are known. This gives a precisely defined class of formulas. Given a theorem T to be proved, we first try to prove it without using induction, using a resolution theorem prover. If this attempt fails, we try to prove the theorem using induction. The first problem to be tackled is to find a suitable induction scheme, i.e. we must discover a suitable well-founded ordering to be used in the application of the principle of induction. Once this has been done, an attempt is made to prove the theorem using this ordering and the induction principle. However, it may happen that this proof fails too, since this theorem may itself depend on another inductive hypothesis or lemma A . Then we have

$$\text{AXIOMS} \wedge A \rightarrow T$$

where “AXIOMS” is the set of axioms required for the proof of this theorem. Therefore

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A$$

i.e. $\neg A$ is a logical consequence of $\text{AXIOMS} \wedge \neg T$. We can therefore use our method for generating logical consequences and unskolemization to derive $\neg A$ from $\text{AXIOMS} \wedge \neg T$. This is a support strategy and will not generate all possible inductive theorems from the axioms, since it makes use of the negation of the theorem T as well as the set of axioms to generate inductive hypotheses. This method can be extended to theorems which depend on more than one inductive hypothesis. The remainder of this section elaborates the ideas outlined above.

5.3.1 Discovering a well-founded ordering

Recall that we have assumed that all well-founded orderings are known (these could be partial as well as total orderings). Now suppose that we are trying to prove the theorem

$$\forall x A(x)$$

where x ranges over some domain D . Consider the set of formulas of the form $A(t)$, where t is a ground term belonging to D and $A(t)$ is first-order provable. We prove some subset of these formulas one by one, noting the proof times for each formula. We denote the time taken to prove $A(t)$ by $PT(t)$. This suggests an ordering in that objects which are smaller in the ordering will probably have smaller proof times. We therefore pick an ordering “ \succ ” such that

$$(X \succ Y) \rightarrow (PT(X) > PT(Y))$$

(at least most of the time).

Example 5.1 Consider the following theorem to be proved by induction :

$$\forall X(\text{reverse}(\text{reverse}(X)) = X),$$

where “*reverse*” is the usual function which reverses lists, and where X ranges over the set of all lists. We first try to prove the theorem for some ground terms using the theorem prover OTTER, a resolution theorem prover developed at the Argonne National Laboratory [McCune 89]. We observe the following proof times for the ground terms given below :

Theorems proved	Time taken (seconds)
$\text{reverse}(\text{reverse}([\])) = [\]$	0.26
$\text{reverse}(\text{reverse}([1])) = [1]$	0.34
$\text{reverse}(\text{reverse}([a])) = [a]$	0.34

$reverse(reverse([2, 1])) = [2, 1]$	1.00
$reverse(reverse([a, b])) = [a, b]$	1.00
$reverse(reverse([3, 2, 1])) = [3, 2, 1]$	2.16
$reverse(reverse([5, 2, 9])) = [5, 2, 9]$	2.16

From the above proof times, the following observations can be drawn :

1. The proof times increase as the length of the list being substituted for X increases; here we have

$$PT([]) < PT([1]) = PT([a]) < PT([2,1]) = PT([a, b]) < PT([3,2,1]) \dots$$

2. The proof times are identical for different ground terms which are lists of the same length; here we have

$$PT([1]) = PT([a]), PT([2,1]) = PT([a, b]), \text{ and so on.}$$

A well-founded ordering \succ which satisfies the condition

$$X \succ Y \rightarrow PT(X) > PT(Y)$$

is therefore the ordering which is defined as follows : for lists X and Y , $X \succ Y$ if and only if the length of list X is greater than the length of list Y ; $X = Y$ if and only if the length of list X is equal to the length of list Y ; and $X \prec Y$ if and only if $Y \succ X$. The minimal element in this ordering is the empty list $[]$. •

5.3.2 Using the induction principle

Once a well-founded ordering W has been found using the method described in the previous section, we must now apply the principle of induction to prove the given theorem $T = \forall x A(x)$. This is done as follows :

Base case : We prove

$$A(m)$$

for all minimal elements m of the ordering W by negating $A(m)$, adding it to the set of axioms, and using resolution to derive the empty clause. If this procedure fails, this means that some induction hypothesis (hypotheses) is (are) required to prove the base case. Sections 5.3.3 – 5.3.4 explain how this case is dealt with.

When the proof of the base case is obtained, we proceed to the induction step.

Induction step : Let $pred(X)$ be the set of elements which precede X in the ordering W . Then we need to prove

$$\forall X \left(\bigwedge_{Y \in pred(X)} A(Y) \rightarrow A(X) \right).$$

Again, this formula is negated, added to the set of axioms and resolution is used to try and derive the empty clause. If the empty clause cannot be thus derived, then as before, one or more induction hypotheses may be required to obtain the proof. This is dealt with in the next two sections.

If a proof is obtained by resolution alone, then the theorem is proved by induction.

Example 5.2 Consider the following theorem :

$$\forall Y \forall Z (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y)))$$

The list notation used here is the same as that used in the programming language LISP. In particular, if X is an element and Y is a list, then $\text{cons}(X, Y)$ is a list of length one more than the length of Y , and contains the element X followed by the list Y . $[]$ or “*nil*” represents the empty list.

Let us try to prove the theorem by induction on Y . Using the same method as in the previous section, we can establish that the same well-founded ordering as that of Example 5.1 can be used here (i.e. for lists X and Y , $X \succ Y$ if and only if the length of list X is greater than the length of list Y ; $X = Y$ if and only if the length of list X is equal to the length of list Y ; and $X \prec Y$ if and only if $Y \succ X$). The minimal element in this ordering is the empty list $[]$, therefore here the base case is :

Base case : $\forall Z (\text{reverse}(\text{append}([], \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}([])))$.

This was proved by OTTER in 0.28 seconds (see the appendix for proof).

Induction step : An element preceding X in the given ordering could be any element of length one less than X . In particular, the list $\text{cdr}(X)$, where $\text{cdr}(X)$ represents the same list as X with the first element removed, is a list with one element less than X . Thus we now need to prove that

$$\forall Y \forall Z ((\text{reverse}(\text{append}(\text{cdr}(Y), \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(\text{cdr}(Y)))) \rightarrow (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y))))$$

This was proved by OTTER in 1.86 seconds (see the appendix for proof).•

5.3.3 Finding one induction hypothesis

Sometimes it may happen that either the base case or the inductive step (or both) of the previous section cannot be proved by resolution alone, i.e. by first-order methods alone. This can occur, for example, if the proof depends on some other lemma which itself needs to be proved by induction before a proof for the actual theorem being proved can be found. Suppose that the proof depends on one lemma A , i.e.

$$\text{AXIOMS} \wedge A \rightarrow T$$

where “AXIOMS” is the set of axioms for the theorem T being proved, and where T can be proved from AXIOMS \wedge A using first-order reasoning. This implication can be rewritten as

$$\neg \text{AXIOMS} \vee \neg A \vee T$$

which is the same as

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A.$$

$\neg A$ can therefore be derived from $(\text{AXIOMS} \wedge \neg T)$ using resolution. To do this, we need to Skolemize $\text{AXIOMS} \wedge \neg T$ before performing resolutions among clauses thus obtained. The clauses which represent $\text{Sk}(\neg A)$ can thus be derived by resolution from $\text{Sk}(\text{AXIOMS} \wedge \neg T)$. We then unskolemize and negate $\text{Sk}(\neg A)$ to obtain the lemma A, using our unskolemization algorithm. Lemma A can now be proved using the methods of Sections 5.3.1 – 5.3.2. Since lemma A has been shown to be valid, and since $\text{AXIOMS} \wedge A \rightarrow T$, the theorem T is also valid, and a proof for T has thus been found.

Example 5.3 We continue with the proof of the theorem of Example 5.1, which was

$$\forall X(\text{reverse}(\text{reverse}(X)) = X).$$

Recall that we found a well-founded ordering W for this theorem in Example 5.1 whose minimal element was the empty list $[\]$. The base step for the proof therefore consists of proving

$$\text{reverse}(\text{reverse}([\])) = [\],$$

which was already done in Example 5.1.

Induction step : An element preceding X in the ordering W could be any element of length one less than X . In particular, the list $\text{cdr}(X)$, where $\text{cdr}(X)$ represents the same list as X with the first element removed, is a list with one element less than X . Thus we need to prove that

$$\forall X((\text{reverse}(\text{reverse}(\text{cdr}(X))) = \text{cdr}(X)) \rightarrow (\text{reverse}(\text{reverse}(X)))).$$

(Call this result T' .) We try to derive the empty clause from $\text{AXIOMS} \wedge \neg T'$, but since T' cannot be proved by induction alone, this attempt fails. However, we succeed in deriving the following clause by resolution from $\text{AXIOMS} \wedge \neg T'$:

$$\forall X(\text{reverse}(\text{append}(\text{reverse}(\text{cdr}(X)), \text{cons}(\text{car}(X), [\]))) \neq \text{cons}(\text{car}(X), \text{reverse}(\text{reverse}(\text{cdr}(X)))).$$

We unskolemize this clause by replacing $\text{reverse}(\text{cdr}(X))$ and $\text{car}(X)$ by new existential variables Y and Z respectively. This yields the formula

$$\exists Y \exists Z(\text{reverse}(\text{append}(Y, \text{cons}(Z, [\]))) \neq \text{cons}(Z, \text{reverse}(Y))).$$

Negating, we obtain the following

Lemma A = $\forall Y \forall Z (\text{reverse}(\text{append}(Y, \text{cons}(Z, []))) = \text{cons}(Z, \text{reverse}(Y)))$.

Lemma A was proved by induction (without the use of any lemmas) in Example 5.2, using the methods of Sections 5.3.1 – 5.3.2. Hence the proof of our theorem

$$\forall X (\text{reverse}(\text{reverse}(X)) = X)$$

is complete. •

5.3.4 Finding more than one induction hypothesis

It may happen that the proof of a theorem T depends on more than one inductive lemma. For example, suppose that inductive lemmas A and B are needed to prove T; in other words, T is first-order derivable from AXIOMS \wedge A \wedge B. Then as before, since

$$\text{AXIOMS} \wedge A \wedge B \rightarrow T,$$

we therefore have

$$\text{AXIOMS} \wedge \neg T \rightarrow \neg A \vee \neg B.$$

Therefore from AXIOMS \wedge $\neg T$, we can derive $\neg A \vee \neg B$ by first-order reasoning. This can be done by resolution from Sk(AXIOMS \wedge $\neg T$). The clauses obtained from these resolutions representing Sk($\neg A \vee \neg B$) can then be unskolemized and negated (as we did in Section 5.3.3 for lemma A). We thus obtain A \wedge B, and each of lemma A and lemma B can be proved by induction using the methods of Sections 5.3.1 – 5.3.2. This method can be extended to any number of inductive lemmas; however, the method rapidly becomes more and more complicated as the number of lemmas increases. For this reason, it may be preferable to use our second approach, described in the next section, for generating inductive hypotheses for such theorems.

5.4 Description of the second method

Suppose that we are trying to prove some theorem T, and suppose that we fail to find a proof of the theorem using standard first-order methods. This suggests that induction may be required to prove the theorem.

If induction is to be used, the first problem is to find a well-founded ordering for the elements. This can be done by the methods described in Section 5.3.1 and this problem will not be further dwelt upon here. If the theorem can now be proved by induction using the well-founded ordering thus discovered, then we are done; if not, then the proof of the theorem may require one or more lemmas, which themselves need to be proved by induction. Our concern in this section is to discover what these lemmas are. If only one lemma is required, then the method of Section 5.3.3 may

prove useful in finding this lemma; however, the following method can be applied to any theorem whose proof requires one or more lemmas, each of which have to be proved by induction. By lemma we mean either some theorem which needs to be proved separately or some instance $T(\bar{Y})$ of the theorem $T(\bar{X})$, where $\bar{Y} < \bar{X}$; such lemmas are also called inductive hypotheses.

Suppose the proof of a theorem T , for which a well-founded ordering P has been discovered, requires n lemmas A_1, A_2, \dots, A_n (these are all unknown). Suppose that T contains m variables ($m \geq 1$), where the i^{th} variable is drawn from some domain D_i for each i , $1 \leq i \leq m$. Let \bar{X} be an m -tuple consisting of these m variables. We write the theorem T as $T(\bar{X})$. Then a proof of the theorem $T(\bar{X})$ will proceed according to the following two steps :

1. Proof of the base case :

The theorem $T(\bar{X})$ is proved to be true for the minimal elements of the ordering P . We will show below that $T(\bar{m})$, for minimal elements \bar{m} of the ordering P , is first-order provable for a certain class of theorems T .

2. Inductive step :

Now the inductive step of the theorem $T(\bar{X})$ is proved. Using lemmas A_1 through A_n as axioms, a proof of $T(\bar{X})$ can be obtained by resolution.

This concludes a proof of $T(\bar{X})$ by induction, using the n lemmas A_1 through A_n as axioms.

Now consider the proof of $T(\bar{Y})$ for some ground element $\bar{Y} \in D_1 \times D_2 \times \dots \times D_m$. This proof can be performed by performing exactly the same steps as in 2 above, except that we now have none of the lemmas A_1 through A_n . As a result of this, ground instances of these lemmas will have to be proved. We will show below that all these ground instances are first-order provable for a certain class of theorems. Thus in this proof of $T(\bar{Y})$, we can find subproofs of lemmas $A_1(\bar{Y})$ through $A_n(\bar{Y})$ ($A_i(\bar{Y})$ denotes the lemma A_i with variables in A_i instantiated to the corresponding values in \bar{Y}).

Since this is true for all ground elements \bar{Y} , the above can be repeated for ground elements $\bar{Y}_1, \bar{Y}_2, \bar{Y}_3, \dots$, and so on. In each proof of $T(\bar{Y}_i)$, we can find subproofs of lemmas $A_1(\bar{Y}_i), A_2(\bar{Y}_i), \dots, A_n(\bar{Y}_i)$.

Now we compare the proofs of $T(\bar{Y})$ for different ground \bar{Y} . These proofs will be similar in structure except that different instances of the lemmas A_1 through A_n will appear in these proofs. By detecting these different instances, we should be able to reconstruct the n lemmas A_1 through A_n . Once these lemmas are known, the theorem $T(\bar{X})$ can be proved by induction using the well-founded ordering P .

In the following theorem, we will show that this method is complete for theorems which can be proved using the usual induction principle, subject to the fol-

lowing restriction. In order to show that this method is complete, we will need to be able to tell whether certain ground terms t which appear in proofs are less than a given ground term z or not. Since ground terms may contain Skolem functions or other functions, it may not always be possible to deduce whether $t < z$ or not. Thus we will only allow those functions t, z for which it is possible to tell whether $t < z$ is true or not. For example, if $z = 3$ and $t = plus(1, 1)$ (where "plus" is the usual addition function for natural numbers), it is possible to deduce that t is less than z ; however, if $z = 3$ and $t = f(5)$ for some Skolem function f , then we cannot tell whether $t < z$ or not.

The feasibility of this method is established in the following theorem.

Theorem 5.1 The method suggested above is complete for theorems that can be proved by first-order logic with the following induction principle:

$$(\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)))$$

where $<$ is a well-founded ordering. Additionally, none of the clauses used should contain terms containing Skolem symbols or other functions for which the question of whether any of these is less than another term is undecidable.

Proof : Suppose we want to prove $P(z)$ for some ground z , where $\forall xP(x)$ is a theorem which can be proved by first-order logic with the above induction principle.

We know that $(\forall x(\forall y(y < x \rightarrow P(y)) \rightarrow P(x)))$ is true. Express this as

$$(\forall x(\exists y)[(y < x \rightarrow P(y)) \rightarrow P(x)]).$$

Substitute the ground term z for x to get

$$(\exists y)[(y < z \rightarrow P(y)) \rightarrow P(z)].$$

Then there are finitely many y_i such that

$$[(y_0 < z \rightarrow P(y_0)) \rightarrow P(z)] \vee \dots \vee [(y_n < z \rightarrow P(y_n)) \rightarrow P(z)]$$

is true. To see this, note that if $(\exists x)A(x)$ is valid for any first order formula $A(x)$, we know there exist finitely many terms t_i such that $A(t_1) \vee \dots \vee A(t_n)$ is valid. We can show this by converting $\neg(\exists x)A(x)$ to clause form with a new predicate for $A(x)$, and looking at the instances of this predicate used in the derivation of the empty clause by resolution.

The above formula can be rewritten as

$$[\neg(y_0 < z \rightarrow P(y_0)) \vee P(z)] \vee \dots \vee [\neg(y_n < z \rightarrow P(y_n)) \vee P(z)]$$

$$\text{i.e. } [\neg(y_0 < z \rightarrow P(y_0)) \vee \dots \vee \neg(y_n < z \rightarrow P(y_n))] \vee P(z)$$

$$\text{i.e. } \neg[(y_0 < z \rightarrow P(y_0)) \wedge \dots \wedge (y_n < z \rightarrow P(y_n))] \vee P(z)$$

$$\text{i.e. } [(y_0 < z \rightarrow P(y_0)) \wedge \dots \wedge (y_n < z \rightarrow P(y_n))] \rightarrow P(z).$$

Now consider any conjunct $(y_i < z \rightarrow P(y_i))$ in the conjunction on the left side of the above implication. If $y_i < z$ is true, then this conjunct is equivalent to

$P(y_i)$; if $y_i < z$ is false, then this conjunct is equivalent to *true* and can therefore be omitted from the above conjunction. We are therefore interested in knowing whether $y_i < z$ is true or false for every i , for $1 \leq i \leq n$. From the assumption in the theorem statement, this question can be answered for all y_i .

Let T be the set of all y_k 's in $\{y_1, y_2, \dots, y_n\}$ such that $y_k < z$. Then

$$\bigwedge_{y \in T} P(y) \rightarrow P(z)$$

Thus $P(z)$ is derivable from a finite conjunction $P(y_{j_1}) \wedge \dots \wedge P(y_{j_q})$ by first-order logic methods, where all the y_{j_i} 's are ground elements less than z .

Repeating the above argument for each of the elements of the set T , we will eventually get

$$P(m_1) \wedge \dots \wedge P(m_s) \rightarrow P(z)$$

where all the m_i 's are minimal elements of the well-founded ordering $<$ (this follows from the fact that $<$ is a well-founded ordering).

Now consider the proof of $P(m_i)$ for some minimal element m_i ($1 \leq i \leq s$). Since m_i is a minimal element, $P(m_i)$ is provable from the given axioms and given lemmas. If the proof of $P(m_i)$ requires the use of lemmas proved by induction previously, then by a simple induction argument on the size of the proof, using the same method as above, we see that we will eventually obtain a first-order proof of $P(m_i)$ from the axioms.

Thus we see that $P(z)$ can be proved by first-order methods; also, this proof is made up of proofs of some $P(y)$'s for $y < z$, which in turn are made up of proofs of some $P(w)$'s for $w < y$, and so on. In other words, each of these proofs have a similar structure, and the theorem is proved. •

A similar theorem can be proved for a slightly different version of the induction principle. In this theorem, no assumptions need to be made regarding the decidability of whether one term is less than another.

Theorem 5.2 The method suggested above is complete for theorems that can be proved by first-order logic with the following induction principle:

If $<$ is a well-founded ordering, and if for all x , $P(x)$ can be proved by first-order logic from the infinite conjunction of $P(y)$ for all $y < x$, then for all x , $P(x)$ is true.

Proof : Suppose we want to prove $P(y)$ for some ground y , where $\forall x P(x)$ is a theorem which can be proved by first-order logic with the above induction principle.

From the assumption in the theorem statement, we know that $P(y)$ can be proved by first-order logic methods from

$$P(y_1) \wedge P(y_2) \wedge \dots \wedge P(y_n) \wedge \dots$$

where $y_i < y \forall i \geq 1$, i.e. $\bigwedge_{i=1}^{\infty} P(y_i) \rightarrow P(y)$.

Therefore by the compactness principle, there exists a finite subset $\{y_{j_1}, y_{j_2}, \dots, y_{j_m}\}$ of the y_i 's such that

$$P(y_{j_1}) \wedge P(y_{j_2}) \wedge \dots \wedge P(y_{j_m}) \rightarrow P(y).$$

Repeating the above argument for each of $P(y_{j_1})$ through $P(y_{j_m})$ in place of $P(y)$, we will eventually get

$$P(m_1) \wedge \dots \wedge P(m_r) \rightarrow P(y)$$

where all the m_i 's are minimal elements of the well-founded ordering $<$ (this follows from the fact that $<$ is a well-founded ordering). As in the proof of Theorem 5.1, since m_i is a minimal element, it is provable from the given axioms and given lemmas. If the proof of $P(m_i)$ requires the use of lemmas proved by induction previously, then by a simple induction argument on the size of the proof, using the same method as above, we see that we will eventually obtain a first-order proof of $P(m_i)$ from the axioms.

Therefore $P(y)$ is also first-order provable. Also, the proof of $P(y)$ can be constructed from the proofs of $P(y_{j_1})$ through $P(y_{j_m})$, each of which in turn can be constructed from a finite conjunction of $P(z_k)$'s, for $z_k < y_{j_k}$ ($1 \leq k \leq m$), and so on. Thus each of these proofs have a similar structure, and the theorem is proved. •

Limitations of this method

1. The first point to note is that for any ground element y , not all proofs of $P(y)$ will have a similar structure to $P(y')$ for other ground elements y' . Potentially, there may exist a large number of different proofs of each ground instance. However, there does exist at least one such proof, as demonstrated in the preceding theorems. We will need to search through the proofs to find one such proof.
2. Given proofs of ground instances of $P(x)$, it is a non-trivial task to detect the similarity in structure between these proofs.

Examples illustrating the use of this method can be found in the appendix.

5.5 Comparison with other methods

The first method given in this chapter uses a support strategy to generate inductive hypotheses from the axioms; the second method is more general and makes use of structural similarities in the proofs of ground instances of the theorem being proved to discover suitable inductive hypotheses. Much of the work in the field

of mechanizing mathematical induction is concentrated in the field of inductionless induction, described in more detail in Section 5.2. As mentioned earlier, inductionless induction applies term rewriting techniques for proving equational theorems. Our methods are more general than these since they can be applied to any theorem which can be proved using the principle of induction. Some more work needs to be done in order to improve the efficiency of our method.

6. Conclusion

6.1 Summary

In this dissertation, we have explored a number of different topics. We first saw how a class of logical consequences of first-order formulas can be derived using resolution and unskolemization. We extended the meaning of “unskolemization” to include replacement of some non-Skolem as well as Skolem functions by existentially quantified variables. This allowed a larger class of logical consequences to be derived, since certain logical consequences of formulas cannot be derived without unskolemization. A detailed algorithm was given to perform this unskolemization, and the properties of formulas derived by applying the algorithm were described.

The remainder of this dissertation revolved around different applications for the above method for deriving logical consequences. We first used the method as part of an algorithm for the automatic generation of loop invariants. The method was applicable since a loop invariant is a logical consequence of the various conditions which are true each time the loop is traversed. We described methods of directing the search for a valid loop invariant and demonstrated their effectiveness with several examples. The algorithm for generating loop invariants in first-order logic was proved to be sound and complete. This is in contrast to all known methods so far, which are heuristics and are by no means complete.

The next topic discussed was machine learning from examples. Given two examples $E1$ and $E2$, a concept learned from $E1$ and $E2$ is a logical consequence of $E1$ and $E2$. Thus we applied our resolution and unskolemization method for deriving logical consequences to this problem. A graph-based algorithm for learning by extracting common features from examples was described, and the properties of the concepts which can be thus learned were discussed. Applications of this learning algorithm to traditional areas such as the blocks world, as well as the mechanical derivation of loop invariants, were demonstrated. The performance and working of our algorithm was compared with those of four other algorithms from the literature, and it was shown that the performance of our algorithm compared favorably with the other four. This work is significant because none of the learning algorithms so

far have used full first-order logic as their representation language. This greatly widens the scope of applicability of our method.

Finally, we described methods for discovering inductive hypotheses for theorems to be proved by induction. Since the principle of mathematical induction is not expressible in first-order logic, in order to be able to prove theorems by induction using only first-order logic, we need to know which inductive hypotheses will be required for the proofs of the theorems. We saw that certain inductive hypotheses can be generated from the axioms and the negation of the theorem by using our method for generating logical consequences. Another method, which involved extracting inductive hypotheses from proofs of ground instances of the theorem, was described. This method was based on the fact that proofs of ground instances of the theorem can have similar structures, and information about which inductive hypotheses are required can be deduced by comparing the structures of these proofs. The method was shown to be complete for certain classes of theorems. It is more general than a large number of existing methods, since it can be applied to equational as well as non-equational theorems. Much of the existing work on this subject deals only with equational theorems.

6.2 Extensions

6.2.1 Automatic generation of loop invariants

We have developed a novel method of automatically deriving loop invariants for flowchart programs. The methods described in this dissertation have not been actually implemented, but have been manually applied to many examples. Many people have voiced the opinion that the goal of automating the derivation of loop invariants is unattainable (see for example [Dijkstra 85]). Of course, they can be proved wrong only if the method we have developed can be made “acceptably” efficient by the use of suitable strategies. Basically, the function GET-APPROX needs to be implemented with the use of strategies which will include rewriting terms to some normal form to improve the efficiency of the resolution procedure, detecting structural similarities among terms, and so on. The function, as it stands now, provides some guidance to the process of deriving the invariants. Its efficiency can probably be greatly improved with the use of some good heuristics. Owing to the existence of a large number of such heuristics in the literature, this aspect has not been explored in much detail here. However, even though heuristics will be able to improve the performance of our algorithm, the algorithm still stands out from the previous purely heuristic methods in the literature. This is because in our

method, heuristics can be embedded within the framework of a complete and sound algorithm. Thus even if all heuristics fail, our algorithm can still derive a correct loop invariant. This is in direct contrast to previously developed methods, which have not been complete in any sense.

Another issue here is that if the given algorithm fails to return a loop invariant for a given program loop, this could be due to one of two reasons : either the invariant is not expressible given the theory axiomatized, or the program is not correct. These two cases cannot be distinguished at present.

6.2.2 Learning from examples

We propose some extensions and modifications to the learning algorithm presented in Chapter 4.

Allowing many-to-many mappings

The algorithm, as presented in Section 4.4.2, performs a one-to-one mapping of arguments from the two given examples. This corresponds to the notion that distinct objects in the two given examples are represented by distinct variables. However, in certain situations it may be desirable to allow different variables to represent the same object. In such a case, it is necessary to allow many-to-many mappings in the argument graph produced by the algorithm. The choice of whether to consider all possible mappings or a limited number of these can be left to the user. A very minor modification to the learning algorithm will allow this feature to be incorporated into the algorithm.

Allowing a limited number of disjunctions

The learning algorithm at present does not allow disjunctions of clauses to be performed if the clauses have no common predicates. If such disjunctions are necessary, they can be permitted, either without restriction or with a limit on the number of disjunctions allowed. This alteration can easily be built into the algorithm.

Using the algorithm for descriptive generalization

The given algorithm provides a method of deriving a formula EX from two given formulas $E1$ and $E2$ such that $E1 \rightarrow EX$, $E2 \rightarrow EX$. However, note that we could also use the algorithm for deriving a formula EX such that $EX \rightarrow E1$,

$EX \rightarrow E2$. To see this, suppose that we are given formulas $E1$ and $E2$; then apply the algorithm to the formulas $\neg E1$ and $\neg E2$. The algorithm produces a formula E such that $\neg E1 \rightarrow E$ and $\neg E2 \rightarrow E$; taking contrapositives, we get $\neg E \rightarrow E1$, $\neg E \rightarrow E2$. Setting $EX = \neg E$, the result follows. In the terminology of Michalski [Michalski 83], this process is known as *descriptive generalization* and is concerned with establishing new concepts or theories characterizing given facts. In this case E_1 and E_2 are the given facts, and EX is the new concept or theory which is established. This method of inference is also known as *abduction* or *abductive inference* [Patterson 90].

6.2.3 Mechanizing mathematical induction

The methods developed in Chapter 5 for generating inductive hypotheses are complete for certain classes of theorems; they need to be made more efficient by the use of suitable strategies. More research needs to be done into ways of detecting structural similarities among proofs of different ground instances of theorems.

Appendix

Time complexity analysis of the learning algorithm

We analyze the algorithm step by step. For convenience, the main body of the algorithm is listed below again, with the steps numbered :

Algorithm LEARN($E_1, E_2, AXIOMS$)

begin

1. Choose $X_c \in Res(E_1 \wedge AXIOMS)$;
2. Choose $Y_c \in Res(E_2 \wedge AXIOMS)$;
3. Rename the variables in all the clauses of X_c and Y_c so that no two clauses have any variable in common;
4. `build_clause_graph`(X_c, Y_c, E_c);
5. `build_argument_graph`(X_a, Y_a, E_a);
6. `augment_graphs_X`(X_a, E_a, X_c, E_c);
7. `augment_graphs_Y`(Y_a, E_a, Y_c, E_c);
8. `maximum_weight_matching`(M_a, X_a, Y_a, E_a);
9. $M_c := \{(C_1, C_2) \in E_c \mid \text{the } n^{\text{th}} \text{ argument } \alpha_n \text{ of some literal of } C_1 \text{ contains } a \text{ as a subterm and the } n^{\text{th}} \text{ argument } \beta_n \text{ of some literal of } C_2 \text{ contains } b \text{ as a subterm in the same position as } a \text{ appears in } \alpha_n \text{ and } (a, b) \in M_a, \text{ for some positive integer } n, \text{ where these two literals have the same predicate}\}$;
10. For every edge $(a, b) \in M_a$ do
 - if $(a$ and b are distinct) and $(a$ and b are not both variables) then
 - replace unmarked occurrences of a and b in M_c by $Z \leftarrow a$ and $Z \leftarrow b$ (respectively) (Z is a new variable);
 - if $(a$ and b are both variables) then
 - unify all occurrences of a and b in M_c ;
11. $EX := \{C_1 \cup C_2 \mid (C_1, C_2) \in M_c\}$;
12. if $EX = \emptyset$ then $EX := true$;
13. for every Skolem function α in EX do
 - if α is not marked then

replace all occurrences of α in all literals of EX by $X \leftarrow \alpha$, where X is a new variable not occurring elsewhere in any clause;

14. Perform Steps 4 through 7 of the unskolemization algorithm for EX ;

end.

We will not analyze the complexity of performing resolutions, since this is a nondeterministic process. The analysis of the algorithm thus begins with step 3 above. The following symbols are used during this analysis :

$|X_c|$: number of elements in the set X_c

$|Y_c|$: number of elements in the set Y_c

$|X_a|$: number of elements in the set X_a

$|Y_a|$: number of elements in the set Y_a

$|C_1|$: maximum cardinality of a clause in X_c

$|C_2|$: maximum cardinality of a clause in Y_c

arg : maximum length of an argument of a clause in $X_c \cup Y_c$ (i.e. maximum number of symbols in an argument; e.g. $f(x, g(y))$ has 4 symbols, viz. f, x, g, y)

$arity$: maximum arity of a predicate in a clause in $X_c \cup Y_c$

$|E_c|$: number of edges of the clause graph (bounded above by $(|X_c| + |Y_c|)^2$)

$|E_a|$: number of edges of the argument graph (bounded above by $(|X_a| + |Y_a|)^2$)

$|M_a|$: size of a maximum weight matching of the argument graph (bounded above by $max(|X_a|, |Y_a|)$)

$|M_c|$: cardinality of the set M_c (bounded above by $|E_c|$)

The maximum number of operations required for each step is given below, within a constant factor.

Step 3 : $|X_c| * |C_1| * arity + |Y_c| * |C_2| * arity$

Step 4 : $|X_c| * |Y_c| * |C_1| * |C_2|$

Step 5 : $|E_c| * |C_1| * arity * |C_2| * arg$

Step 6 : $|X_a| * |Y_a| * |Y_a| + (|X_a| + |Y_a|) * (|X_c| + |E_c|)$

Step 7 : $|Y_a| * |X_a| * |X_a| + (|X_a| + |Y_a|) * (|Y_c| + |E_c|)$

Step 8 : $|E_a| * (|X_a| + |Y_a|) * \log_{\lceil |E_a| / (|X_a| + |Y_a|) + 1 \rceil} (|X_a| + |Y_a|)$

Step 9 : $|E_c| * |M_a| * |C_1| * |C_2| * arity * arg$

Step 10 : $|M_a| * |M_c| * (|C_1| + |C_2|) * arity * arg$

Step 11 : $|M_c|$

Step 12 : *constant*

Step 13 : $|M_c| * arg * (|C_1| + |C_2|) * arity$

Step 14 : Each of steps 4 through 7 of the unskolemization algorithm take time $|M_c| * (|C_1| + |C_2|) * arity$, for the generation of one unskolemized formula. •

Working of problem from Section 4.7 using the algorithm LEARN

We process the examples in the order E_1, E_2 and E_3 . The examples are shown in Figure 4.10. The same result is obtained for other orders of presentation of the examples.

First the examples E_1 and E_2 are taken and all possible resolutions are performed between these two examples and the given axioms. The resulting sets of clauses obtained from E_1 and E_2 , called X_c and Y_c respectively, are :

$$X_c = medium(a) \wedge polygon(a) \wedge blank(a) \wedge ontop(a, b) \wedge medium(b) \wedge circle(b) \wedge shaded(b) \wedge ontop(b, c) \wedge large(c) \wedge polygon(c) \wedge blank(c),$$

$$Y_c = medium(d) \wedge polygon(d) \wedge blank(d) \wedge ontop(d, e) \wedge small(f) \wedge circle(f) \wedge shaded(f) \wedge inside(f, e) \wedge small(g) \wedge circle(g) \wedge shaded(g) \wedge inside(g, e) \wedge large(e) \wedge polygon(e) \wedge blank(e).$$

We build the clause and argument graphs for X_c and Y_c ; these graphs are shown in Figures A.1 and A.2 respectively. These graphs do not need to be augmented since neither X_c nor Y_c contain any variables. There exists two maximum weight matchings for the argument graph; these are

$$\{(a, d)(4), (b, f)(2), (c, e)(4)\} \text{ and } \{(a, d)(4), (b, g)(2), (c, e)(4)\}$$

(the weights for each edge are indicated after each edge in parentheses). However, it turns out that both these matchings give rise to the same concept. We therefore choose the first matching and get

$$M_a = \{(a, d), (b, f), (c, e)\}.$$

The set M_c contains the edges which are shown in Figure A.3. We then replace a and d by $X \leftarrow a$ and $X \leftarrow d$ respectively; we replace b and f by $Y \leftarrow b$ and $Y \leftarrow f$ respectively; and we replace c and e by $Z \leftarrow c$ and $Z \leftarrow e$ respectively in the edges of M_c . We then get

$$EX = \{\{medium(X \leftarrow a), medium(X \leftarrow d)\}, \{polygon(X \leftarrow a), polygon(X \leftarrow d)\}, \{blank(X \leftarrow a), blank(X \leftarrow d)\}, \{ontop(X \leftarrow a, Y \leftarrow b), ontop(X \leftarrow d, Z \leftarrow e)\}, \{circle(Y \leftarrow b), circle(Y \leftarrow f)\}, \{shaded(Y \leftarrow b), shaded(Y \leftarrow f)\}, \{ontop(Y \leftarrow b, Z \leftarrow c), ontop(X \leftarrow d, Z \leftarrow e)\}, \{large(Z \leftarrow c), large(Z \leftarrow e)\}, \{blank(Z \leftarrow c), blank(Z \leftarrow e)\}, \{polygon(Z \leftarrow c), polygon(Z \leftarrow e)\}\}.$$

We now unskolemize EX by replacing the marked arguments by existentially quantified variables and get

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge circle(Y) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z) \wedge polygon(Z)).$$

This is the concept learned from E_1 and E_2 . We will now apply the algorithm to E_3 and the above formula EX . First we perform all possible resolutions between the axioms and these examples. EX remains unchanged; we get the following set of clauses from E_3 :

$$E_3 = \{\{medium(h)\}, \{polygon(h)\}, \{blank(h)\}, \{ontop(h, j)\}, \{medium(j)\}, \{polygon(j)\}, \{shaded(j)\}, \{ontop(j, k)\}, \{large(k)\}, \{ellipse(k)\}, \{blank(k)\}\}.$$

We now need to express EX in clause form. After Skolemizing EX , we get the set of clauses

$$\{\{medium(s)\}, \{polygon(s)\}, \{blank(s)\}, \{ontop(s, t), ontop(s, u)\}, \{circle(t)\}, \{shaded(t)\}, \{ontop(t, u), ontop(s, u)\}, \{large(u)\}, \{blank(u)\}\},$$

where s, t, u are Skolem functions replacing the existentially quantified variables X, Y, Z respectively.

We build the clause and argument graphs for these two sets of clauses; these graphs are shown in Figures A.4 and A.5 respectively. These graphs do not need to be augmented since none of the clauses contain any variables. The maximum weight matching for the argument graph is :

$$M_a = \{(h, s)(5), (j, t)(3), (k, u)(5)\}$$

(the weights for each edge are indicated after each edge in parentheses). The set M_c contains the edges which are shown in Figure A.6. We then replace h and s by $X \leftarrow h$ and $X \leftarrow s$ respectively; we replace j and t by $Y \leftarrow j$ and $Y \leftarrow t$ respectively; and we replace k and u by $Z \leftarrow k$ and $Z \leftarrow u$ respectively in the edges of M_c . We then get

$$EX = \{\{medium(X \leftarrow h), medium(X \leftarrow s)\}, \{polygon(X \leftarrow h), polygon(X \leftarrow s)\}, \{blank(X \leftarrow h), blank(X \leftarrow s)\}, \{ontop(X \leftarrow h, Y \leftarrow j), ontop(X \leftarrow s, Y \leftarrow t), ontop(X \leftarrow s, Z \leftarrow u)\}, \{ontop(X \leftarrow h, Y \leftarrow j), ontop(Y \leftarrow t, Z \leftarrow u), ontop(X \leftarrow s, Z \leftarrow u)\}, \{shaded(Y \leftarrow j), shaded(Y \leftarrow t)\}, \{ontop(Y \leftarrow j, Z \leftarrow k), ontop(Y \leftarrow t, Z \leftarrow u), ontop(X \leftarrow s, Z \leftarrow u)\}, \{ontop(Y \leftarrow j, Z \leftarrow k), ontop(X \leftarrow s, Y \leftarrow t), ontop(X \leftarrow s, Z \leftarrow u)\}, \{large(Z \leftarrow k), large(Z \leftarrow u)\}, \{blank(Z \leftarrow k), blank(Z \leftarrow u)\}\}.$$

We now unskolemize EX by replacing the marked arguments by existentially quantified variables and get

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge (ontop(X, Y) \vee ontop(Y, Z) \vee ontop(X, Z)) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z)).$$

This is the concept learned from E_1 , E_2 and E_3 . Note that one of the disjunctions here is subsumed by two of the others, namely the disjunction $(ontop(X, Y) \vee ontop(Y, Z) \vee ontop(X, Z))$; therefore it can be discarded. The resulting concept learned from the three given examples is

$$EX = \exists X \exists Y \exists Z (medium(X) \wedge polygon(X) \wedge blank(X) \wedge (ontop(X, Y) \vee ontop(X, Z)) \wedge shaded(Y) \wedge (ontop(Y, Z) \vee ontop(X, Z)) \wedge large(Z) \wedge blank(Z)). \bullet$$

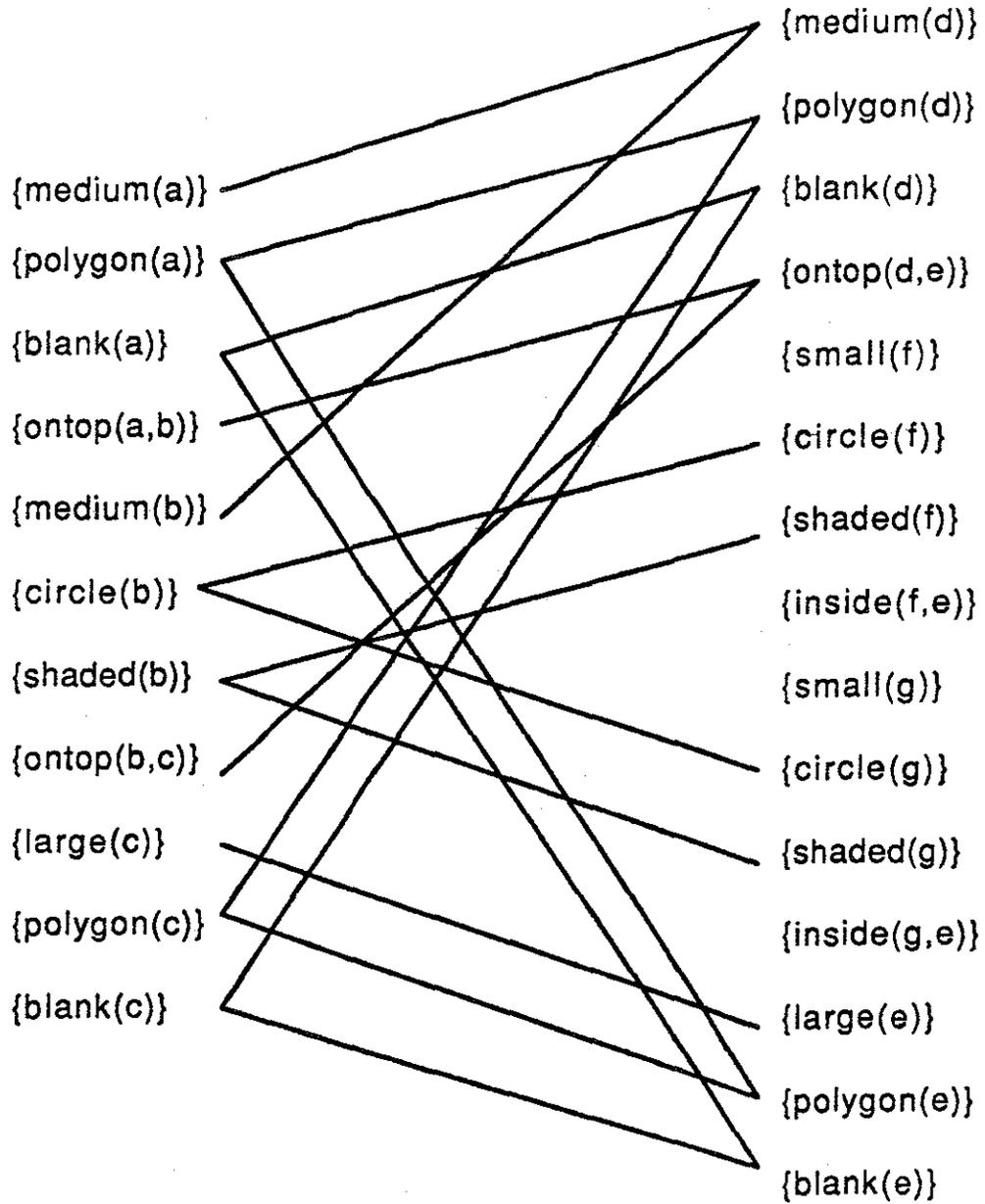


Figure A.1 Clause graph for E_1 and E_2

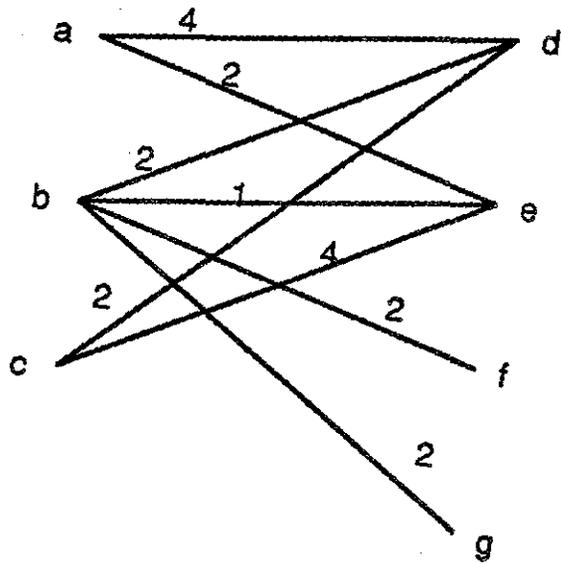


Figure A.2 Argument graph for E_1 and E_2

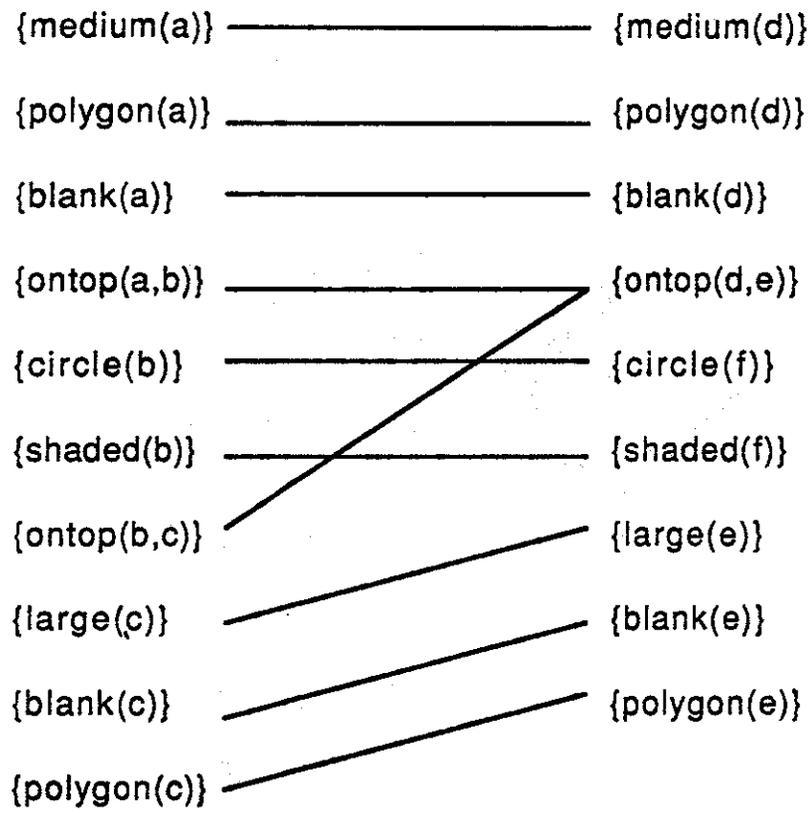


Figure A.3 Subgraph of clause graph for E_1 and E_2

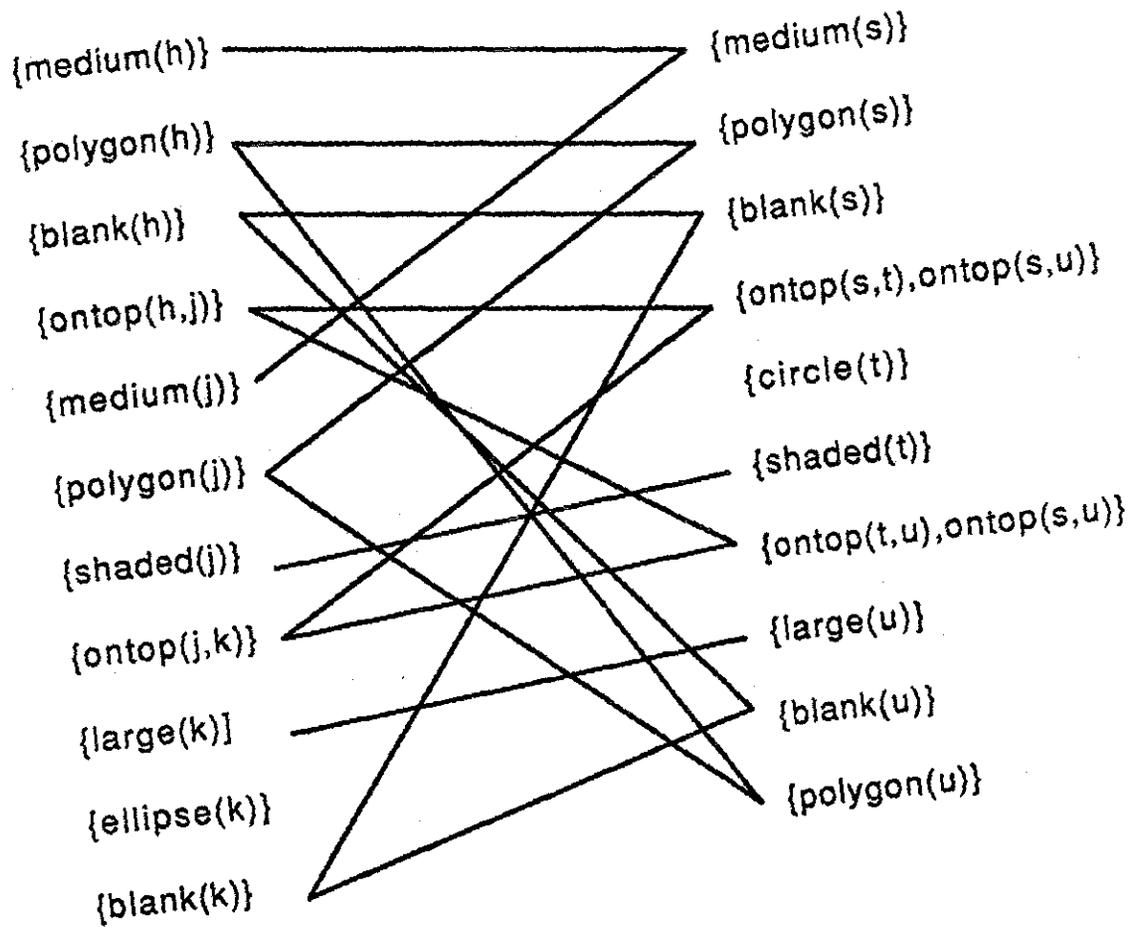


Figure A.4 Clause graph for E_3 and EX

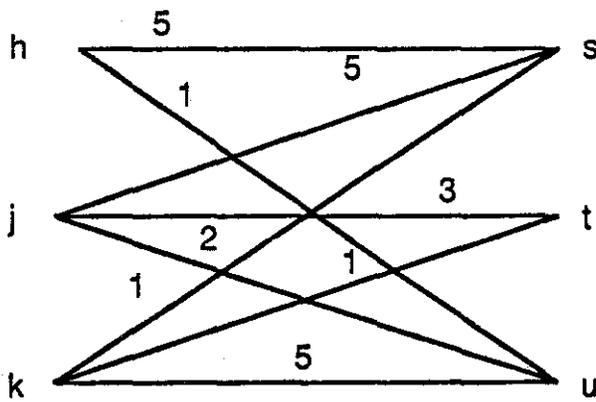


Figure A.5 Argument graph for E_3 and EX

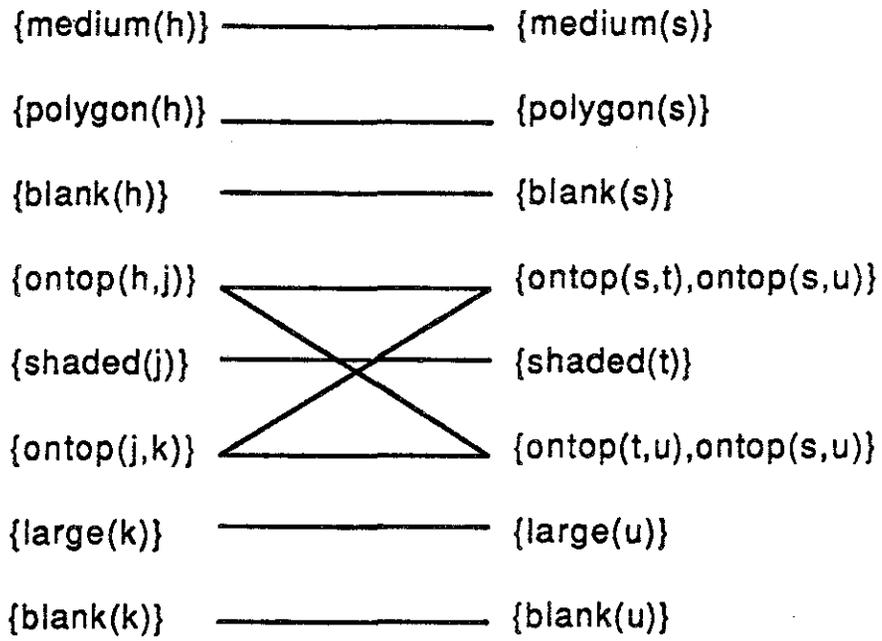


Figure A.6 Subgraph of clause graph for E_3 and EX

Proofs for Example 5.2

The following proofs were obtained for Example 5.2 using the theorem prover OTTER :

Proof of base step from Example 5.2 :

- 1 [] (X = X).
- 2 [] (append(nil,Y) = Y).
- 7 [] (reverse(append(nil,cons(y,nil))) != cons(y,reverse(nil))).
- 8 [] (reverse(nil) = nil).
- 11 [] (reverse(cons(X,nil)) = cons(X,nil)).
- 12 [para_into,7,2,demod,11,8] (cons(y,nil) != cons(y,nil)).
- 13 [binary,12,1] .

Proof of induction step from Example 5.2 :

- 1 [] (X = X).
- 3 [] (append(X,Y) = cons(car(X),append(cdr(X),Y))) | -listp(X).
- 4 [] (reverse(X) = append(reverse(cdr(X)),cons(car(X),nil))) | -listp(X).
- 5 [] listp(cons(X,Y)).
- 7 [] (reverse(append(cdr(x),cons(y,nil))) = cons(y,reverse(cdr(x)))).
- 8 [] listp(x).
- 9 [] (reverse(append(x,cons(y,nil))) != cons(y,reverse(x))).
- 11 [] (car(cons(X,Y)) = X).
- 12 [] (cdr(cons(X,Y)) = Y).
- 14 [para_into,9,3] (reverse(cons(car(x), append(cdr(x), cons(y, nil)))) != cons(y, reverse(x))) | -listp(x).
- 17 [binary,14,8] (reverse(cons(car(x), append(cdr(x),cons(y, nil)))) != cons(y, reverse(x))).
- 19 [para_into,17,4,demod,12,11] (append(reverse(append(cdr(x), cons(y,nil))), cons(car(x), nil)) != cons(y, reverse(x))) | -listp(cons(car(x), append(cdr(x), cons(y, nil))))).
- 54 [binary,19,5] (append(reverse(append(cdr(x), cons(y, nil))),cons(car(x), nil)) != cons(y, reverse(x))).
- 56 [para_into,54,7] (append(cons(y, reverse(cdr(x))), cons(car(x), nil)) != cons(y, reverse(x))).
- 61 [para_into,56,3,demod,11,12] (cons(y, append(reverse(cdr(x)), cons(car(x), nil))) != cons (y, reverse(x))) | -listp(cons(y, reverse(cdr(x)))).
- 63 [binary,61,5] (cons(y, append(reverse(cdr(x)), cons(car(x), nil))) != cons(y, reverse(x))).

65 [para.into,63,4] (cons(y,reverse(x)) != cons(y,reverse(x))) | -listp(x).
 68 [binary,65,1] -listp(x).
 69 [binary,68,8]. •

Working of examples for Section 5.4

We give below three examples illustrating the technique outlined in Section 5.4 for discovering inductive hypotheses.

Example 5.4 To illustrate the discussion in Section 5.4, suppose that we are trying to prove the commutativity of addition, using the Peano axioms :

1. $\forall x((x = 0) \vee (x = a + 1))$ where a is a Skolem symbol
2. $\forall x \forall y((x \neq y + 1) \vee (x \neq 0))$
3. $\forall x \forall y((x + 1 \neq y + 1) \vee (x = y))$
4. $\forall x \forall y(\neg(x < y + 1) \vee (x < y) \vee (x = y))$
5. $\forall x \forall y(\neg(x < y) \vee (x < y + 1))$
6. $\forall x \forall y((x \neq y) \vee (x < y + 1))$
7. $\forall x(\neg(x < 0))$
8. $\forall x \forall y((x < y) \vee (x = y) \vee (y < x))$
9. $\forall x(x + 0 = x)$
10. $\forall x \forall y(x + (y + 1) = (x + y) + 1)$
11. $\forall x(x * 0 = 0)$
12. $\forall x \forall y(x * (y + 1) = x * y + x)$
13. $\forall x(x = x)$

The theorem to be proved is

$$\forall x \forall y(x + y = y + x)$$

An attempt to prove this theorem without induction, using only the above axioms and resolution, fails. We therefore start trying to prove ground instances of the theorem. Three ground proofs are shown below :

1) **Proof of $(1+1) + (((1+1)+1)+1) = (((1+1)+1)+1) + (1+1)$.**

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{(1 + 1) + (((1 + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$

1. $\{(((1 + 1) + ((1 + 1) + 1)) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1))\}$

paramodulate with axiom 10

2. $\{(((1 + 1) + (1 + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$

- paramodulate with axiom 10
3. $\{(((((1 + 1) + 1) + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 4. $\{(((1 + 1) + 1) + (1 + 1)) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 5. $\{((((1 + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 6. $\{(((1 + 1) + 1) + 1) + (1 + 1) \neq (((1 + 1) + 1) + 1) + (1 + 1)\}$
paramodulate with axiom 10
 7. $\{\}$ resolve with axiom 13

2) Proof of $(1+1) + (((1+1)+1)+1)+1 = (((1+1)+1)+1)+1 + (1+1)$.

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{(1+1)+(((1+1)+1)+1)+1 \neq (((1+1)+1)+1)+1+(1+1)\}$

1. $\{(1 + 1) + (((1 + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
2. $\{((1 + 1) + (((1 + 1) + 1) + 1)) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
3. $\{(((1 + 1) + ((1 + 1) + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
4. $\{((((1 + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
5. $\{((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
6. $\{((((((1 + 1) + 1) + (1 + 1)) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
7. $\{(((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
8. $\{((((((1 + 1) + 1) + 1) + (1 + 1)) + 1) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
9. $\{((((((1 + 1) + 1) + 1) + 1) + (1 + 1)) + 1) + 1 \neq (((1 + 1) + 1) + 1) + 1 + (1 + 1)\}$
paramodulate with axiom 10
10. $\{\}$ resolve with axiom 13.

3) Proof of $((1+1)+1) + (((1+1)+1)+1)+1 = (((1+1)+1)+1)+1 + ((1+1) + 1)$.

We use paramodulation as well as resolution as inference rules and obtain the following refutation proof of the negation of the theorem :

Negation of theorem : $\{((1 + 1) + 1) + (((1 + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$

1. $\{(((1 + 1) + 1) + ((1 + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
2. $\{((((1 + 1) + 1) + ((1 + 1) + 1)) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
3. $\{((((((1 + 1) + 1) + (1 + 1)) + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
4. $\{(((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
5. $\{((((((1 + 1) + 1) + 1) + (1 + 1)) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
6. $\{((((((1 + 1) + 1) + 1) + ((1 + 1) + 1)) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
7. $\{(((((((1 + 1) + 1) + 1) + (1 + 1)) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
8. $\{((((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
9. $\{(((((((1 + 1) + 1) + 1) + 1) + (1 + 1)) + 1) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
10. $\{(((((((1 + 1) + 1) + 1) + 1) + ((1 + 1) + 1)) \neq (((1 + 1) + 1) + 1) + ((1 + 1) + 1)\}$
paramodulate with axiom 10
11. $\{\}$ resolve with axiom 13.

It can be seen that each proof contains an instance of the proof of the lemma

$$\forall x \forall y ((y - 1) + x) + 1 = ((y - 1) + 1) + x,$$

namely in clauses 4, 8, and 6 for the three proofs respectively.

This lemma can easily be proved by induction, using the well-founded order $<$. The given theorem can then be proved by induction, using this lemma as an axiom. •

Example 5.5 Let us try to prove the theorem

$$\forall x \forall y (x * y = y * x).$$

An attempt to prove this theorem by first-order methods, using the Peano axioms given in Example 5.4, fails. We therefore try to prove ground instances of the theorem. We assume that the following simple theorem has already been

proved : $\forall x(0 + x = x)$. Some proofs of ground instances of the theorem are given below :

1) Proof of $1*(1+1) = (1+1)*1$.

Negation of the theorem : $1 * (1 + 1) \neq (1 + 1) * 1$.

The proof proceeds as follows :

1. $\{(1 * 1) + 1 \neq (1 + 1) * 1\}$
paramodulate with axiom 12
2. $\{(1 * (0 + 1)) + 1 \neq (1 + 1) * 1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
3. $\{((1 * 0) + 1) + 1 \neq (1 + 1) * 1\}$
paramodulate with axiom 12
4. $\{(0 + 1) + 1 \neq (1 + 1) * 1\}$
paramodulate with axiom 11
5. $\{0 + (1 + 1) \neq (1 + 1) * 1\}$
paramodulate with axiom 10
6. $\{((1 + 1) * 0) + (1 + 1) \neq (1 + 1) * 1\}$
paramodulate with axiom 11
7. $\{(1 + 1) * (0 + 1) \neq (1 + 1) * 1\}$
paramodulate with axiom 12
8. $\{(1 + 1) * 1 \neq (1 + 1) * 1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
9. $\{\}$ paramodulate with axiom 13.

2) Proof of $1*((1+1)+1) = ((1+1)+1)*1$.

Negation of the theorem : $1 * ((1 + 1) + 1) \neq ((1 + 1) + 1) * 1$.

The proof proceeds as follows :

1. $\{(1 * (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with axiom 12
2. $\{((1 * 1) + 1) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with axiom 12
3. $\{((1 * (0 + 1)) + 1) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with Theorem $\forall x(0 + x = x)$
4. $\{(((1 * 0) + 1) + 1) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with axiom 12
5. $\{((0 + 1) + 1) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with axiom 11
6. $\{(0 + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
paramodulate with axiom 10
7. $\{(((1 + 1) * 0) + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$

- paramodulate with axiom 11
8. $\{((1 + 1) * (0 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
9. $\{((1 + 1) * 1) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
10. $\{((1 + 1) * (0 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
11. $\{(((1 + 1) * 0) + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
12. $\{(0 + (1 + 1)) + 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 11
13. $\{0 + ((1 + 1) + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 10
14. $\{(((1 + 1) + 1) * 0) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 11
15. $\{((1 + 1) + 1) * (0 + 1) \neq ((1 + 1) + 1) * 1\}$
 paramodulate with axiom 12
16. $\{((1 + 1) + 1) * 1 \neq ((1 + 1) + 1) * 1\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
17. $\{\}$ resolve with axiom 13

3) Proof of $(1+1)*((1+1)+1) = ((1+1)+1)*(1+1)$.

Negation of the theorem : $(1 + 1) * ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)$.

The proof proceeds as follows :

1. $\{((1 + 1) * (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
2. $\{(((1 + 1) * 1) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
3. $\{(((1 + 1) * (0 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
4. $\{((((1 + 1) * 0) + (1 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
5. $\{(((0 + (1 + 1)) + (1 + 1)) + (1 + 1)) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 11
6. $\{(((1 + 1)) + (1 + 1)) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
7. $\{(((1 + 1) + 1) + 1) + (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 10
8. $\{(((1 + 1) + 1) + (1 + 1)) + 1 \neq ((1 + 1) + 1) * (1 + 1)\}$

- paramodulate with axiom 10
9. $\{((1 + 1) + 1) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 10
10. $\{(0 + ((1 + 1) + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
11. $\{((((1 + 1) + 1) * 0) + ((1 + 1) + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 11
12. $\{(((1 + 1) + 1) * (0 + 1)) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
13. $\{(((1 + 1) + 1) * 1) + ((1 + 1) + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with Theorem $\forall x(0 + x = x)$
14. $\{((1 + 1) + 1) * (1 + 1) \neq ((1 + 1) + 1) * (1 + 1)\}$
 paramodulate with axiom 12
15. $\{\}$ resolve with axiom 13.

From the proofs of the three above ground instances, we see that each proof contains a subproof of the lemma

$$\forall x \forall y ((x + 1) * y = (x * y) + y),$$

namely in clauses 1, 9, and 1 respectively of the three proofs.

And this lemma can be proved by induction, using the well-founded order $<$. The given theorem can then be proved by induction, using this lemma as an axiom. •

Example 5.6 In this example, we solve the same problem as that solved in Example 5.3 in Chapter 5, i.e. we are trying to prove the theorem

$$\forall x(\text{reverse}(\text{reverse}(x)) = x)$$

by induction, this time using the method described in this section. A well-founded ordering for this example was already discovered in Example 5.1 in Chapter 5.

We prove the theorem for different values of ground x :

Proof of $\text{reverse}(\text{reverse}([a, b])) = [a, b]$:

The axioms used in this proof are (in clause form) :

1. $X = X$
2. $\text{append}(\text{nil}, Y) = Y$
3. $\text{append}(X, Y) = \text{cons}(\text{car}(X), \text{append}(\text{cdr}(X), Y)) \vee \neg \text{listp}(X)$
4. $\text{reverse}(\text{nil}) = \text{nil}$
5. $\text{reverse}(X) = \text{append}(\text{reverse}(\text{cdr}(X)), \text{cons}(\text{car}(X), \text{nil})) \vee \neg \text{listp}(X)$
6. $\text{car}(\text{cons}(X, Y)) = X$
7. $\text{cdr}(\text{cons}(X, Y)) = Y$

8. $listp(cons(X, Y))$

Negation of theorem :

$$reverse(reverse([a, b])) \neq [a, b]$$

Using a set of support strategy, we get the following proof for the theorem :

9. $reverse(reverse([a, b])) \neq [a, b]$

negation of theorem

10. $reverse(append(reverse(cdr([a, b])), cons(car([a, b]), nil))) \neq [a, b],$

$\neg listp([a, b])$

paramodulate 9,5

11. $reverse(append(reverse(cdr([a, b])), cons(car([a, b]), nil))) \neq [a, b]$

resolve 8,10

12. $reverse(append(reverse([b]), cons(car([a, b]), nil))) \neq [a, b]$

paramodulate 7,11

13. $reverse(append(reverse([b]), [a])) \neq [a, b]$

paramodulate 6,12

14. $reverse(append(append(reverse(cdr([b])), cons(car([b]), nil)), [a])) \neq [a, b],$

$\neg listp([b])$

paramodulate 5,13

15. $reverse(append(append(reverse(cdr([b])), [b]), [a])) \neq [a, b], \neg listp([b])$

paramodulate 6,14

16. $reverse(append(append(reverse(nil), [b]), [a])) \neq [a, b], \neg listp([b])$

paramodulate 7,15

17. $reverse(append(append(reverse(nil), [b]), [a])) \neq [a, b]$

resolve 8,16

18. $reverse(append(append(nil, [b]), [a])) \neq [a, b]$

paramodulate 4,17

19. $reverse(append([b], [a])) \neq [a, b]$

paramodulate 2,18

20. $reverse(cons(car([b]), append(cdr([b]), [a]))) \neq [a, b], \neg listp([b])$

paramodulate 3,19

21. $reverse(cons(car([b]), append(nil, [a]))) \neq [a, b], \neg listp([b])$

paramodulate 7,20

22. $reverse(cons(car([b]), append(nil, [a]))) \neq [a, b]$

resolve 8,21

23. $reverse(cons(car([b]), [a])) \neq [a, b]$

paramodulate 2,22

24. $reverse([b, a]) \neq [a, b]$

- paramodulate 6,23
25. $append(reverse(cdr([b, a])), cons(car(cons(b, [a]))) \neq [a, b], \neg listp([b])$
 paramodulate 5,24
26. $append(reverse([a]), cons(car([b, a]), nil)) \neq [a, b], \neg listp([b])$
 paramodulate 7,25
27. $append(reverse([a]), [b]) \neq [a, b], \neg listp([b])$
 paramodulate 6,26
28. $append(reverse([a]), [b]) \neq [a, b]$
 resolve 8,27
29. $append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]) \neq [a, b], \neg listp([a])$
 paramodulate 5,28
30. $append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]) \neq [a, b]$
 resolve 8,29
31. $append(append(reverse(nil), cons(car([a]), nil)), [b]) \neq [a, b]$
 paramodulate 7,30
32. $append(append(reverse(nil), [a]), [b]) \neq [a, b]$
 paramodulate 6,31
33. $append(append(nil, [a]), [b]) \neq [a, b]$
 paramodulate 4,32
34. $append([a], [b]) \neq [a, b]$
 paramodulate 2,33
35. $cons(car([a]), append(cdr([a]), [b])) \neq [a, b], \neg listp([a])$
 paramodulate 3,34
36. $cons(car([a]), append(cdr([a]), [b])) \neq [a, b]$
 resolve 8,35
37. $cons(a, append(cdr([a]), [b])) \neq [a, b]$
 paramodulate 6,36
38. $cons(a, append(nil, [b])) \neq [a, b]$
 paramodulate 7,37
39. $[a, b] \neq [a, b]$
 paramodulate 2,38
40. empty clause
 resolve 1,39.

Proof of $reverse(reverse([a, b, c])) = [a, b, c]$:

Using a set of support strategy, and the same axioms (1 through 8 above), we get the following proof for the theorem :

9. $reverse(reverse([a, b, c])) \neq [a, b, c]$

negation of theorem

10. $reverse(append(reverse(cdr([a, b, c])), cons(car([a, b, c]), nil))) \neq [a, b, c],$
 $\neg listp([a, b, c])$
paramodulate 5,9
11. $reverse(append(reverse([b, c]), cons(car([a, b, c]), nil))) \neq [a, b, c],$
 $\neg listp([a, b, c])$
paramodulate 7,10
12. $reverse(append(reverse([b, c]), [a])) \neq [a, b, c], \neg listp([a, b, c])$
paramodulate 6,11
13. $reverse(append(reverse([b, c]), [a])) \neq [a, b, c]$
resolve 8,12
14. $reverse(append(append(reverse(cdr([b, c])), cons(car([b, c]), nil)), [a])) \neq$
 $[a, b, c], \neg listp([b, c])$
paramodulate 5,13
15. $reverse(append(append(append(reverse(cdr([b, c])), cons(car([b, c]), nil)), [a])) \neq$
 $[a, b, c]$
resolve 8,14
16. $reverse(append(append(reverse([c]), cons(car([b, c]), nil)), [a])) \neq [a, b, c]$
paramodulate 7,15
17. $reverse(append(append(reverse([c]), [b]), [a])) \neq [a, b, c]$
paramodulate 6,16
18. $reverse(append(append(append(reverse(cdr([c])), cons(car([c]), nil)), [b]), [a]))$
 $\neq [a, b, c], \neg listp([c])$
paramodulate 5,17
19. $reverse(append(append(append(reverse(cdr([c])), [c]), [b]), [a])) \neq [a, b, c],$
 $\neg listp([c])$
paramodulate 6,18
20. $reverse(append(append(append(reverse(nil), [c]), [b]), [a])) \neq [a, b, c],$
 $\neg listp([c])$
paramodulate 7,19
21. $reverse(append(append(append(reverse(nil), [c]), [b]), [a])) \neq [a, b, c]$
resolve 8,20
22. $reverse(append(append(append(nil, [c]), [b]), [a])) \neq [a, b, c]$
paramodulate 4,21
23. $reverse(append(append([c], [b]), [a])) \neq [a, b, c]$
paramodulate 2,22
24. $reverse(append(cons(car([c]), append(cdr([c]), [b])), [a])) \neq [a, b, c], \neg listp([c])$
paramodulate 3,23

25. $reverse(append(cons(car([c]), append(cdr([c]), [b])), [a])) \neq [a, b, c]$
 resolve 8,24
26. $reverse(append(cons(c, append(cdr([c]), [b])), [a])) \neq [a, b, c]$
 paramodulate 6,25
27. $reverse(append(cons(c, append(nil, [b])), [a])) \neq [a, b, c]$
 paramodulate 7,26
28. $reverse(append([c, b], [a])) \neq [a, b, c]$
 paramodulate 2,27
29. $reverse(cons(car([c, b]), append(cdr([c, b]), [a]))) \neq [a, b, c], \neg listp([c, b])$
 paramodulate 3,28
30. $reverse(cons(car([c, b]), append(cdr([c, b]), [a]))) \neq [a, b, c]$
 resolve 8,29
31. $reverse(cons(c, append(cdr([c, b]), [a]))) \neq [a, b, c]$
 paramodulate 6,30
32. $reverse(cons(c, append([b], [a]))) \neq [a, b, c]$
 paramodulate 7,31
33. $reverse(cons(c, cons(car([b]), append(cdr([b]), [a]))) \neq [a, b, c], \neg listp([b])$
 paramodulate 3,32
34. $reverse(cons(c, cons(car([b]), append(cdr([b]), [a]))) \neq [a, b, c]$
 resolve 8,33
35. $reverse(cons(c, cons(b, append(cdr([b]), [a]))) \neq [a, b, c]$
 paramodulate 6,34
36. $reverse(cons(c, cons(b, append(nil, [a]))) \neq [a, b, c]$
 paramodulate 7,35
37. $reverse([c, b, a]) \neq [a, b, c]$
 paramodulate 2,36
38. $append(reverse(cdr([c, b, a])), cons(car([c, b, a]), nil)) \neq [a, b, c], \neg listp([c, b, a])$
 paramodulate 5,37
39. $append(reverse(cdr([c, b, a])), cons(car([c, b, a]), nil)) \neq [a, b, c]$
 resolve 8,38
40. $append(reverse([b, a]), cons(car([c, b, a]), nil)) \neq [a, b, c]$
 paramodulate 7,39
41. $append(reverse([b, a]), [c]) \neq [a, b, c]$
 paramodulate 6,40
42. $append(append(reverse(cdr([b, a])), cons(car([b, a]), nil)), [c]) \neq [a, b, c],$
 $\neg listp([b, a])$
 paramodulate 5,41
43. $append(append(reverse([a]), cons(car([b, a]), nil)), [c]) \neq [a, b, c], \neg listp([b, a])$

- paramodulate 7,42
44. $append(append(reverse([a]), [b]), [c]) \neq [a, b, c], \neg listp([b, a])$
paramodulate 6,43
45. $append(append(reverse([a]), [b]), [c]) \neq [a, b, c]$
resolve 8,44
46. $append(append(append(reverse(cdr([a])), cons(car([a]), nil)), [b]), [c])$
 $\neq [a, b, c], \neg listp([a])$
paramodulate 5,45
47. $append(append(append(reverse(nil), cons(car([a]), nil)), [b]), [c]) \neq [a, b, c],$
 $\neg listp([a])$
paramodulate 7,46
48. $append(append(append(reverse(nil), [a]), [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 6,47
49. $append(append(append(nil, [a]), [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 4,48
50. $append(append([a], [b]), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 2,49
51. $append(append([a], [b]), [c]) \neq [a, b, c]$
resolve 8,50
52. $append(cons(car([a]), append(cdr([a]), [b])), [c]) \neq [a, b, c], \neg listp([a])$
paramodulate 3,51
53. $append(cons(car([a]), append(cdr([a]), [b])), [c]) \neq [a, b, c]$
resolve 8,52
54. $append(cons(a, append(cdr([a]), [b])), [c]) \neq [a, b, c]$
paramodulate 6,53
55. $append(cons(a, append(nil, [b])), [c]) \neq [a, b, c]$
paramodulate 7,54
56. $append([a, b], [c]) \neq [a, b, c]$
paramodulate 2,55
57. $cons(car([a, b]), append(cdr([a, b]), [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 3,56
58. $cons(a, append(cdr([a, b]), [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 6,57
59. $cons(a, append([b], [c])) \neq [a, b, c], \neg listp([a, b])$
paramodulate 7,58
60. $cons(a, append([b], [c])) \neq [a, b, c]$
resolve 8,59
61. $cons(a, cons(car([b]), append(cdr([b]), [c]))) \neq [a, b, c], \neg listp([b])$

- paramodulate 3,60
62. $\text{cons}(a, \text{cons}(b, \text{append}(\text{cdr}([b]), [c]))) \neq [a, b, c], \neg \text{listp}([b])$
 paramodulate 6,61
63. $\text{cons}(a, \text{cons}(b, \text{append}(\text{nil}, [c]))) \neq [a, b, c], \neg \text{listp}([b])$
 paramodulate 7,62
64. $\text{cons}(a, \text{cons}(b, \text{append}(\text{nil}, [c]))) \neq [a, b, c]$
 resolve 8,63
65. $[a, b, c] \neq [a, b, c]$
 paramodulate 2,64
66. empty clause
 resolve 1,65.

From the above two ground proofs, it can be seen that instances of the lemma $\forall x \forall y (\text{reverse}(\text{append}(x, \text{cons}(y, \text{nil}))) = \text{cons}(y, \text{reverse}(x)))$ were proved in both proofs. This is a lemma which needs to be proved by induction and was proved in Example 5.2 in Chapter 5 earlier. The given theorem can then be proved by induction, using this lemma as an axiom. •

References

- Angluin, Dana, Smith, Carl H. : "Inductive Inference : Theory and Methods", *Computing Surveys* 15 (3), pp. 237-269 (1983).
- Aubin, Raymond : "Mechanizing Structural Induction Part I : Formal System", *Theoretical Computer Science* 9, pp. 329-345 (1979).
- Aubin, Raymond : "Mechanizing Structural Induction Part II : Strategies", *Theoretical Computer Science* 9, pp. 347-362 (1979).
- Biundo, S., Hummel, B., Hutter, D., Walther, C. : "The Karlsruhe Induction Theorem Proving System", *Eighth International Conference on Automated Deduction*, pp. 672-674 (1986).
- Bondy, J. A., Murty, U. S. R. : "Graph Theory with Applications", *Elsevier Science Publishing Co., Inc., New York* (1976).
- Boyer, Robert S., Moore, J. Strother : "A Computational Logic", *Academic Press, Inc., New York* (1979).
- Buchanan, B. G., Mitchell, Tom M. : "Model-directed learning of production rules", *Proceedings of the Workshop on Pattern-directed Inference Systems, Honolulu, Hawaii* (1977).
- Burstall, R. M. : "Proving properties of programs by structural induction", *Computer Journal* 12 (1), pp. 41-48 (1969).
- Caplain, Michel, "Finding invariant assertions for proving programs", *Proceedings of the International Conference on Reliable Software*, pp. 165-171 (1975).
- Chang, Chin-Lian, Lee, Richard Char-Tung, "Symbolic Logic and Mechanical Theorem Proving", *Academic Press Inc., New York* (1973).
- Cook, Stephen A., "Soundness and completeness of an axiom system for program verification", *SIAM Journal on Computing* 7 (1), pp. 70-90 (1978).
- Cooper, D. C., "Programs for Mechanical Program Verification", *Machine Intelligence* 6, pp. 43-59 (1971).
- Cox, P. T., Pietrzykowski, T., "A complete, nonredundant algorithm for reversed skolemization", *Theoretical Computer Science* 28, pp. 239-261 (1984).
- Davis, M., Putnam, H., Robinson, J., "The decision problem for exponential Diophantine equations", *Annals of Mathematics* 74, pp. 425-436 (1961).
- Deutsch, Laurence P., "An Interactive Program Verifier", *Ph.D. dissertation, University of California at Berkeley* (1973).

- Dietterich, Thomas G., Michalski, Ryszard S : "A Comparative Review of Selected Methods for Learning from Examples", in : *Machine Learning : An Artificial Intelligence Approach*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 41-82 (1983).
- Dijkstra, E. W., "Invariance and non-determinacy", *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, pp. 157-165 (1985).
- Dijkstra, E. W., "On the cruelty of really teaching computing science", *Communications of the ACM* 32 (12), pp. 1398-1404 (1989).
- Elsplas, Bernard, Levitt, Karl N., Waldinger, Richard J., Waksman, Abraham, "An Assessment of Techniques for Proving Program Correctness", *ACM Computing Surveys* 4 (2), pp. 97-147 (1972).
- Floyd, R. W., "Assigning meanings to programs", *Proceedings of the Symposium on Applied Mathematics*, American Mathematical Society 19, pp. 19-32 (1967).
- Galil, Zvi, "Efficient Algorithms for Finding Maximum Matching in Graphs", *ACM Computing Surveys* 18 (1), pp. 23-38 (1986).
- German, Steven M., Wegbreit, Ben, "A Synthesizer of Inductive Assertions", *IEEE Transactions on Software Engg.*, Vol. SE-1 (1), pp. 68-75 (1975).
- Goguen, J. A. : "How to prove algebraic inductive hypotheses without induction", *Fifth International Conference on Automated Deduction*, pp. 356-373 (1980).
- Gold, E. Mark : "Language Identification in the Limit", *Information and Control* 10, pp. 447-474 (1967).
- Good, Donald I., London, Ralph L., Bledsoe, W. W., "An Interactive Program Verification System", *IEEE Transactions on Software Engg.*, Vol. SE-1 (1) (1975).
- Good, Donald I., Cohen, R. M., Hoch, C. G., Hunter, L. W., Hare, D. F., "Report on the language Gypsy, Version 2.0", *Technical Report ICSCA-CMP-10, Certifiable Minicomputer, Project, ICSCA, The University of Texas at Austin* (1978).
- Good, Donald I., "Mechanical proofs about computer programs", *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson eds., Prentice-Hall, pp. 55-75 (1985).
- Gries, David, "The Science of Programming", Springer-Verlag (1981).
- Hayes-Roth, Frederick, McDermott, John : "An Interference Matching Technique for Inducing Abstractions", *Communications of the ACM* 21 (5), pp. 401-410 (1978).
- Hoare, C.A.R., "An axiomatic basis for computer programming", *Communications of the ACM* 12, pp. 576-580 (1969).
- von Henke, F. W., Luckham, D. C., "A methodology for verifying programs", *Proceedings of the International Conference on Reliable Software*, pp. 156-164 (1975).

- Huet, G, Hullot, J.M. : "Proofs by induction in equational theories with constructors", *Journal of Computer and System Sciences* 25 (2) (1982).
- Jouannaud, Jean-Pierre, Kounalis, Emmanuel : "Automatic Proofs by Induction in Equational Theories Without Constructors", *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pp. 358-366 (1986).
- Kapur, Deepak, Musser, David R. : "Inductive reasoning with incomplete specifications" *Proceedings of the Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pp. 367-377 (1986).
- Katz, Shmuel M., Manna, Zohar, "A heuristic approach to program verification", *Third International Joint Conference on Artificial Intelligence*, pp. 500-512 (1973).
- King, James C., "A Program Verifier", *Ph.D. dissertation, Carnegie-Mellon University* (1969).
- King, James C., "An Interpretation Oriented Theorem Prover over Integers", *Second Annual ACM Symposium on Theory of Computing*, pp. 169-179 (1970).
- King, James C., "Proving Programs to be Correct", *IEEE Transactions on Computers*, Vol. C-20 (11), pp. 1331-1336 (1970).
- Knuth, D., Bendix, P. : "Simple Word Problems in Universal Algebras", in *Computational Problems in Abstract Algebra*, ed. J. Leech, Pergamon Press, pp. 263-297 (1970).
- Kodratoff, Y., Ganascia, J. G., Clavieras, B., Bollinger, T., Tecuci, G. : "Careful generalization for concept learning", in *Advances in Artificial Intelligence*, T. O'Shea (Ed.), Elsevier Science Publishers B.V. (North-Holland), pp. 229-238 (1985).
- Kodratoff, Y., Ganascia, J. G. : "Improving the Generalization Step in Learning", in : *Machine Learning : An Artificial Intelligence Approach*, Vol. II, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 215-244 (1986).
- Lee, Shie-Jue : "CLIN : An Automated Reasoning System Using Clause Linking", *Ph.D. Dissertation, University of North Carolina, Chapel Hill* (1990).
- Lewis, Harry R., Papadimitriou, Christos H., "Elements of the theory of computation", *New Jersey : Prentice-Hall, Inc.* (1981).
- Loeckx, Jacques, Sieber, Kurt, "The Foundations of Program Verification", *John Wiley and Sons, Ltd.* (1987).
- Loveland, Donald, "Automated Theorem Proving , A Logical Basis", *North-Holland Publishing Co.* (1978).
- Manna, Zohar, "Second-order Mathematical Theory of Computation", *Second Annual ACM Symposium on Theory of Computing*, pp. 158-168 (1970).
- Manna, Zohar, Ness, Stephen, Vuillemin, Jean : "Inductive Methods for Proving Properties of Programs", *Communications of the ACM* 16 (8), pp. 491-502 (1973).

- Manna, Zohar, "Mathematical Theory of Computation", McGraw-Hill, New York (1974).
- McCarthy, John : "A basis for a mathematical theory of computation", in *Computer Programming and Formal Systems*, eds. P. Braffort, D. Hirschberg, North-Holland Publishing Co., pp. 33-70 (1970).
- McCune, William W., "Un-Skolemizing clause sets", *Information Processing Letters* **29**, pp. 257-263 (1988).
- McCune, William W., "OTTER 1.0 Users' Guide", *Computer Science Division, Argonne National Laboratory, Argonne, Illinois* (1989).
- Michalski, Ryszard S. : "Learning by inductive inference", *Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-74-671* (1974).
- Michalski, Ryszard S. : "Toward Computer-Aided Induction : A brief review of currently implemented AQVAL programs", *Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-77-874* (1977).
- Michalski, Ryszard S. : "A Theory and Methodology of Inductive Learning", *Machine Learning : An Artificial Intelligence Approach, Vol. I*, eds. R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 83-134 (1983).
- Michalski, Ryszard S. : "Machine Learning : An Artificial Intelligence Approach", *Vol. II*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc. (1986).
- De Millo, R. A., Lipton, R. J., Perlis, A. J., "Social processes and proofs of theorems and programs", *Communications of the ACM* **22**, pp. 271-280 (1979).
- Mitchell, Tom M. : "Version Spaces : A candidate elimination approach to rule learning", *Fifth International Joint Conference on Artificial Intelligence, MIT, Cambridge MA, Vol. 1*, pp. 305-310 (1977).
- Mitchell, Tom M. : "Learning and Problem Solving", *Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, Germany*, pp. 1139-1151 (1983).
- Musser, David R. : "On proving inductive properties of abstract data types", *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 154-162 (1980).
- Nakajima, R., Yuasa, T., "The IOTA programming system", *Lecture Notes in Computer Science* **160**, Springer-Verlag (1983).
- Nelson, Greg, Oppen, Derek C., "Simplification by Cooperating Decision Procedures", *ACM Transactions on Programming Languages and Systems* **1** (2), pp. 245-257 (1979).
- Patterson, Dan W. : "Introduction to Artificial Intelligence and Expert Systems", Prentice Hall, Englewood Cliffs, New Jersey (1990).

- Paul, E. : "Proof by induction in equational theories with relations between constructors" *Ninth Colloquium on Trees in Algebra and Programming, Bordeaux, France*, ed. B. Courcelle, pp. 211-225 (1984).
- Polak, Wolfgang, "Compiler Specification and Verification", *Lecture Notes in Computer Science 124*, Springer - Verlag (1981).
- Robinson, J. A., "A Machine-oriented Logic based on the Resolution Principle", *Journal of the ACM 12 (1)*, pp. 23-41 (1965).
- Rulifson, J. F., Waldinger, R. J., Derksen, J., "A language for writing problem-solving programs", *IFIP Cong. 1971, Yugoslavia, North-Holland Publ. Co., Amsterdam (1972)*.
- Sammut, Claude, Banerji, Ranan B. : "Learning Concepts by Asking Questions", in : *Machine Learning : An Artificial Intelligence Approach, Vol. II*, eds. R.S. Michalski, J.G. Carbonell, T.M. Mitchell, Morgan Kaufmann Publishers, Inc., pp. 167-192 (1986).
- Sarkar, D., De Sarkar, S. C., "Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers", *IEEE Transactions on Software Engineering 15 (1)*, pp. 1-9 (1989).
- Sarkar, D., De Sarkar, S. C., "A Set of Inference Rules for Quantified Formula Handling and Array Handling in Verification of Programs over Integers", *IEEE Transactions on Software Engineering 15 (11)*, pp. 1368-1381 (1989).
- Sarkar, D., De Sarkar, S. C., "A Theorem Prover for Verifying Iterative Programs Over Integers", *IEEE Transactions on Software Engineering 15 (12)*, pp. 1550-1566 (1989).
- Shapiro, Ehud Y. : "An Algorithm that Infers Theories from Facts", *Seventh International Joint Conference on Artificial Intelligence*, pp. 446-451 (1981).
- Spitzen, Jay, Wegbreit, Ben, "The Verification and Synthesis of Data Structures", *Acta Informatica 4*, pp. 127-144 (1975).
- Stanford Verification Group, "Stanford Pascal Verifier User Manual", *Stanford Verification Group Report No. 11 (1979)*.
- Stepp, R. : "The investigation of the UNICLASS inductive program AQ7UNI and User's Guide", *Technical Report 949, Department of Computer Science, University of Illinois, Urbana, Illinois (1978)*.
- Suzuki, Narihasa, "Verifying programs by Algebraic and Logical Reduction", *Proceedings of the International Conference on Reliable Software*, pp. 473-481 (1975).
- Tinkham, Nancy : "Induction of Schemata for Program Synthesis", *Ph.D. dissertation, Department of Computer Science, Duke University, Durham N.C. (1990)*.
- Toyama, Y. : "How to prove equivalence of term rewriting systems without induction", *Eighth International Conference on Automated Deduction*, pp. 118-127 (1986).

- Utgoff, Paul E. : "Machine Learning of Inductive Bias", *Kluwer Academic Publishers, Massachusetts* (1986).
- Valiant, L. G. : "A Theory of the Learnable", *Communications of the ACM* 27 (11), pp. 1134-1142 (1984).
- Vanlehn, Kurt : "Efficient Specialization of Relational Concepts", *Machine Learning* 4, pp. 99-106 (1989).
- Vere, Steven A. : "Induction of Concepts in the Predicate Calculus", *Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia USSR, Vol. 1*, pp. 281-287 (1975).
- Vere, Steven A. : "Inductive learning of relational productions", in : *Pattern-Directed Inference Systems*, eds. D.A. Waterman, Frederick Hayes-Roth, Academic Press, Inc., New York, pp. 281-295 (1978).
- Wand, M., "A new incompleteness result for Hoare's system", *Journal of the ACM* 25, pp. 168-175 (1978).
- Watanabe, Larry, Rendell, Larry, "Effective Generalization of Relational Descriptions", *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 875-881 (1990).
- Wegbreit, Ben, "Heuristic Methods for Mechanically Deriving Inductive Assertions", *Proceedings of Third International Joint Conference on Artificial Intelligence* (1973).
- Wegbreit, Ben, Spitzen, Jay M. : "Proving Properties of Complex Data Structures", *Journal of the ACM* 23 (2), pp. 389-396 (1976).
- Winston, P. H. : "Learning structural descriptions from examples", in : *The Psychology of Computer Vision*, ed. P. H. Winston, McGraw-Hill, New York, pp. 157-209 (1975).
- Zhang, Hantao, Kapur, Deepak, Krishnamoorthy, Mukkai S. : "A mechanizable induction principle for equational specifications", *Ninth International Conference on Automated Deduction, Argonne, Illinois*, pp. 162-181 (1988).