A novel term rewriting strategy for certain hierarchical AC-algebraic systems

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Additional Information:

- A Doctoral Thesis. Submitted in partial fulfillment of the requirements for the award of Doctor of Philosophy of Loughborough University.

Metadata Record: https://dspace.lboro.ac.uk/2134/10641

Publisher: © Ifeyinwa Okoli

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/
<table>
<thead>
<tr>
<th>AUTHOR/FILING TITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>OKOLI, I</td>
</tr>
</tbody>
</table>

|  |  |  |  |  |  |  |  |  |  |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
| VOL. NO.          | CLASS MARK        | LOAN COPY         |
|  |  |  |  |  |  |  |  |  |  |
Library
University of Technology
LOUGHBOROUGH
Leicestershire. LE11 3TU

DESPATCH ADDRESS

-5.SEP89.00
A NOVEL TERM REWRITING
STRATEGY FOR CERTAIN HIERARCHICAL
AC-ALGEBRAIC SYSTEMS

by

Ifeyinwa Okoli

A Doctoral Thesis
Submitted in partial fulfilment of the requirements
for the award of Doctor of Philosophy
of the Loughborough University of Technology
August, 1989.

Supervisor: Doctor D. J. Cooke.
Department of Computer Studies.

© by Ifeyinwa Okoli, 1989.
DECLARATION

I declare that the following thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

IFY OKOLI
In memory of my father,
Fredrick Dim,
May your soul rest in perfect peace,
Amen.
ACKNOWLEDGEMENTS

I wish to give warm thanks to my supervisor, Dr. D.J. Cooke, for his advice and readiness to help even with my personal problems.

My sincere gratitude goes to the Federal Government of Nigeria, for her financial support.

Thanks are also due to my employer, University of Maiduguri, for providing the opportunity for me to benefit from their in-service training programme.

May I also take this opportunity to show my sincere appreciation to the assistance given to me by some of the staff of the department, namely: Dr. R.G. Stone, Mr. G.S. Samra, Mrs. J. Poulton and Dr. C.J. Hinde.

My heartfelt thanks also go to Dr. A. Benson for sacrificing his precious time to proof read my thesis.

Last but not the least, my thanks to my Director of Research, Prof. E.A. Edmonds, for his guidance.
CONTENTS

Abstract.

Chapter 1. INTRODUCTION

Chapter 2. TERM-REWRITING SYSTEMS

2.1 Automated Deduction
2.2 Formal Systems
   2.2.1 Definitions and Notations
   2.2.2 Sequent Calculus
   2.2.3 A Refutational system
   2.2.4 Resolution Systems
2.3 Term-rewriting Methods
   2.3.1 Definitions and Notations
   2.3.2 Orderings
      2.3.2.1 Orderings Based on Operator Precedence
   2.3.3 Term-rewriting Modulo Equations
   2.3.4 Completion
   2.3.5 Narrowing
2.4 Comparison of Simplification Orderings
2.5 Conclusion
Chapter 3. DERIVATION OF HIERARCHICAL OPERATOR ORDERING

3.1 Pre-requisites
3.1.1 Grammatical Pre-requisites
3.1.2 Derivation of Operator Ordering
    3.1.2.1 Standard Forms
3.2 Presentation of Hierarchical Operator Ordering
3.3 Exploitation of Hierarchical Operator Ordering to Orientate Equations
    3.3.1 Orientation of Equations
3.4 Conclusion

Chapter 4. A NEW STRATEGY FOR TERM-REWITING SYSTEMS I:
AN EXAMPLE

4.1 A New Approach
4.2 Illustration of the Strategy as Applied to a Boolean Algebra
    4.2.1 The Orientation of Equations
4.2.2 The Normalization Procedure
    4.2.2.1 The Initial Expression
    4.2.2.2 Adjusting Operator Ordering
    4.2.2.3 Restructuring and Simplifying of Expression Using AC Properties of Operators
    4.2.2.4 Standardization of Sub-expression Length
    4.2.2.5 Factorization of The PDNF
4.3 Conclusion
Chapter 5. A NEW STRATEGY FOR TERM-REWIRTING SYSTEMS II:

FORMAL PRESENTATION

5.1 The Method
5.1.1 Operator Re-ordering
5.1.2 Layer Simplification
5.1.3 Sub-expressions Standardization
5.1.4 Factorization

5.2 Formal Requirements
5.2.1 Operators
5.2.2 Rewrite Rules
  5.2.2.1 Operator Re-ordering
  5.2.2.2 Layer Simplification
  5.2.2.3 Sub-expressions Standardization
  5.2.2.4 Factorization

5.3 Conclusion

Chapter 6. ON THE INADEQUACY OF EQUATIONS

6.1 On The Operator Ordering of The Target Form
  6.1.1 Operator Ordering Inadequacy
  6.1.2 Operator Set Inadequacy
  6.1.3 Inadequacy of The Target Form

6.2 Insufficient Sets of Equations
6.3 Non-AC Operations
6.4 Conclusion
Chapter 7. PRACTICAL CONSIDERATIONS

7.1 Algebraic Specification
7.2 Construction of Forms From Sets of Operators
7.3 Representation of Binary Relations as Boolean matrices
   7.3.1 Implementation
7.4 Computation of The Closure of The Matrix
   7.4.1 Interpretation of The Closure Matrix
      7.4.1.1 Relationship Between Closure Matrix and Rewriting
7.5 Incomplete Algebraic Systems
7.6 Space Considerations
   7.6.1 Characteristics of Traditional "In-situ" Closure Algorithm
   7.6.2 Complement Form
7.7 Conclusion

Chapter 8. COMBINED ALGEBRAIC SYSTEMS

8.1 Introduction of New Operators
8.2 Combined Rewriting Systems
8.3 Matrix Calculations
8.4 Conclusion
Chapter 9. APPLICATIONS OF TERM-REWRITING

9.1 In Specification Languages
   9.1.1 The OBJ Language
   9.1.2 Realization of Function Definitions
   9.1.3 Functional Parallel Programming

9.2 As a Program Verification Tool

9.3 Program Transformation

9.4 Conclusion

Chapter 10. CONCLUSION

References
ABSTRACT

We demonstrate a new strategy for term rewriting in which associative-commutative (AC) operators are used to partition an expression into layers which can be processed separately. The AC-property of the principal operator at each level then allows for local reorganization into a form which is consistent with the "natural" ordering of identifiers. Whilst this representation is easy to process (by hand or by machine) it can also be very lengthy. Therefore, a more condensed representation is sought by "ordered factorization", the ordering being dictated by lexicographical considerations. This necessitates the re-application of some rules in the reverse direction - a distinctive feature of our strategy.

There is a problem of the sufficiency in Term Rewriting Systems (TRSs) - the ability of a TRS to guarantee "normalization" of every expression in the algebra. To characterize this phenomenon we introduce the triple: (algebra (A), set of rules (R), set of expressions (E)). Each time A and R are fixed E, the set of expressions which R can normalize, needs to be ascertained. R is complete if and only if E equates to all expressions derivable within A.

Matrix calculations are employed in the investigation of the completeness of R. The method presented gives a partial solution to the problem and in the case of "failure" it highlights the "algebraic" gaps within R.
Extensions of the basic technique to deal with mixed sorts are suggested. Here a mixture of operators is present and classically all their defining equations are presented in one set. Application of our matrix analysis seeks to partition such a set in ways consistent with our stratified hierarchy. Partitioning the equation set significantly reduces the effort required to normalize expressions in these more complex algebraic systems.

The application of term-rewriting strategies to program transformations is also mentioned. Although little can be said about the topic at this time, it seems to be a potentially important area worth serious investigation.
1. INTRODUCTION

Term rewriting can be regarded as a theorem proving strategy, i.e. a logical statement is said to be proven, if it is a logical expression and it reduces to the value true. To illustrate this we consider a proof of the expression:

\[ x \lor (x \land y) = x \land (x \lor y) \]

By applying the distribution of \( \lor \) over \( \land \) to the lhs we obtain the expression:

\[ (x \lor x) \land (x \lor y) = x \land (x \lor y) \]

Applying the idempotent rule: \( x \lor x \rightarrow x \), to the appropriate subterm of the lhs we have the expression:

\[ x \land (x \lor y) = x \land (x \lor y) \]

Since the both sides are identical the logical expression:

true

is produced, therefore the original equality holds.

Term-re rewriting may also be used as a computational technique, i.e. replacing equals by equals in an evaluation process. For example, the methodology may be used to replace some subterms of an expression by their (possible) values. The evaluation is completed when all the subterms have been replaced by their values. In particular, in the integer expression:

\[ 2 \times 3 + 4 \times 5 \]

the subterm: \( 2 \times 3 \) can be replaced by the product, 6, to give:

\[ 6 + 4 \times 5 \]

In the same manner, the subterm, \( 4 \times 5 \), is replaced by 20, and the resulting subterm:
$6+20$

is finally replaced by

$26$

which is the result of the evaluation.

If a term rewriting system (TRS) is given as a set of equations, there is an inherent problem of non-termination of the rewriting process. This is due to the symmetry of the equality relation. For example, let us consider an equation $2 = r$ and an expression: $s + 2 + t$. Using the equation the expression is rewritten as: $s + r + t$. Again, using the same equation in the opposite direction the original expression is obtained; and this can go on for ever. To avoid this situation, the equations are oriented to form what are called rewrite rules. They indicate the appropriate directions the equations are applied in an attempt to ensure termination. However, we shall show that if the rewriting process is segmented, equations can be usefully applied in different directions in different segments. This can yield more 'sensible' results without introducing non-termination.

The classical TRS of Knuth-Bendix and its subsequent extensions treated expressions from an algebraic point of view; we want to approach the problem from a linguistic stand-point rather than purely algebraically.

Chapter 2 gives a brief "potted history" of term-rewriting. This includes the general background, and an introduction to orderings and completion procedures to be found in the
literature. There is also mention of the comparison of some orderings employed in proofs of termination in some TRSs.

We derive, in chapter 3, an ordering for orienting equations that drive our TRS. The ordering is based on grammars in which a hierarchy of the operator symbols is strictly observed. The application of the ordering is also illustrated.

In chapter 4, we introduce a new strategy for TRS. Our new approach is basically the segmentation of rewriting. This uses the associative-commutative property of certain operator symbols which give stratified "standard" forms. The existence of these forms then enables the development of a system based on a sequence of sub-systems. A description of a system in which the strategy is utilized for manipulating a Boolean algebra, is then given. Each segment is discussed in detail and illustrated with examples.

Chapter 5 gives a formal description of general case and identifies sufficient conditions for the successful termination of the process when applied to a single-sorted algebraic system.

In chapter 6, we investigate the likely causes of an inadequacy of the term rewriting strategy. These are, in most cases, due to the ordering on the terminal symbols and on the lack of suitable rewrite rules. Some simple
illustrations are presented. Also included is a description of how non-AC operations can be handled.

We deliberate, in chapter 7, on the practical considerations associated with our notion of completeness. A set of equations (rewrite rules) is used to derive a relation which is then coded as a binary matrix. Some calculations are performed on the matrix and the results establish the completeness/incompleteness of a given set of equations modulo a given operator ordering. A digression on efficiency considerations is also included.

Chapter 8 demonstrates the extension of our strategy to handle a composite algebraic system. Composite systems are constituted of different sorts, operations and identities. We therefore need to obtain a homogeneous system. Also, to handle the resulting system more efficiently, the rewriting needs to be done in stages, each dealing with only a small set of rules. This is consistent with the single-sorted case and is beneficial in that the methodology reduces a complex problem to a progression of less complex ones.

Chapter 9 outlines some areas of software engineering where term-rewriting strategies have been successfully applied. We indicate the possible application of the methodology in a new area - program transformation.
In chapter 10 we summarize what we have done, point out some limitations and ways in which the methodology might be taken further.
2. TERM-REWRITING SYSTEMS

In this chapter we present a brief history of earlier work which lead to the development of the term-rewriting. This is followed by an overview of the achievements by some individuals in the field. Our interest will focus on the so-called path orderings which can be employed in the investigation of the termination of some term-rewriting systems (TRSs) and completion strategies. Also included is the comparison of these path orderings and the ordering proposed by Knuth and Bendix which was based on weightings.

2.1 Automated Deduction

Term-rewriting evolved from early work on 'automated deduction'. Automated deduction in a narrow sense is the proof of mathematical theorems with the aid of computer. The ideal situation, i.e. complete automation, can be represented diagrammatically as in figure 2.1:

![Figure 2.1](image)

Figure 2.1 shows the input of a theorem into a theorem prover (an algorithm) which executes the theorem and produces a
'yes' if the theorem is true and a 'no' otherwise. In the broad sense, automated deduction is problem solving in a mathematical or logical language, i.e. in a formal system.

Gödel (1931) (Cf. [DKMSH 86])

In the 1920's, Hilbert proposed that the structure of a language, the definitions and all of the mathematical axioms and principles of logic used in developing a suitable selected portion of the existing classical mathematics, should be completely specified in terms of just the forms of the objects of the language. In other words, that the portion in question could be formalized or embedded in a formal system, FS. He also proposed that, by mathematical reasoning on the most elementary and intuitive level, it should be possible to prove that a FS is (simply) consistent, i.e. that no two sequences of formulae exist of which one is a proof of a formula, $A$, and the other the proof of its negation, $\neg A$, in the same FS.

Gödel set out to prove the consistency of analysis, in pursuance of Hilbert's programme. He reduced the consistency analysis to that of number theory. He considered a model in which the set of variables are interpreted as ranging over sets of definable arithmetic values. He soon realized that he would need not only the consistency of number theory but also its truth. A formal analog of this can be used to infer the truth in number theory cannot be defined in number theory.
According to Hilbert, a domain of mathematics could be formalized in a FS. It is therefore required that a FS should include everything that is necessary for proving (true) propositions about that domain. Also, since, either \(A\) or \(\neg A\) is provable in FS, then FS will be (simply) complete.

Gödel showed that when the usual number theory is taken as the domain, there is a proposition \(A\) which can be shown to be formally undecidable, i.e. neither the formula \(A\) nor the negation \(\neg A\) is provable in the FS. This lead to his incompleteness theorem of 1931. See also [Man 66] and [Rog 67].

2.2 Formal Systems

These are systems involving an artificial language in which the notions of 'meaningful expression', axioms and inference rules are precisely described.

2.2.1 Definitions and Notations

The negation symbol is denoted by \(\neg\).

A term is a variable or a string of symbols consisting of a function symbol of degree \(n \geq 0\) followed by \(n\) terms.

An atomic formula \((A)\) is a string consisting of a predicate symbol of degree \(n \geq 0\) followed by \(n\) terms.

A literal \((L)\) is an atomic formula; also if \(A\) is an atomic formula then \(\neg A\) is a literal.

Two literals \(L\) and \(\neg L\) are said to be each other's complement. The pair \((L, \neg L)\) is called a complementary pair.
A clause \((C)\) is a disjunction of literals, i.e.
\[
C = L_1 \lor L_2 \lor \ldots \lor L_n.
\]
A ground literal is a literal that does not contain variables.
A ground clause is a clause in which every literal is ground.

A well-formed formula (wff) is any string of symbols which qualifies to be either a term or a literal.
Axioms are a special set of wffs.
Inference rules are a set of relations among wffs.

A model is a disjunction of literals which does not include a complementary pair.
A sentence \((S)\) is a conjunction of clauses, denoted as:
\[
S = C_1 \land C_2 \land \ldots \land C_n.
\]
A sentence \(S\) is \textit{satisfiable} if there is a model of \(S\); otherwise \(S\) is \textit{unsatisfiable}.

If \(LVC\) and \(\neg LVD\) are two ground Clauses, then the ground clause \(CVD\), is called a \textit{ground resolvent} of \(LVC\) and \(\neg LVD\).
If \(S\) is a sentence, then the \textit{ground resolution} of \(S\), is the sentence consisting of the members of \(S\) together with all the ground resolvents of all pairs of clauses in \(S\). An example is given in the later section.

A substitution \(\Psi\) is a mapping from a set of variables \(V\) to a set of terms \(T\), where \(V\) and the set of variables in \(T\) are disjoint.
If $E$ is a wff, and $\Psi$ is a substitution, then the **instantiation** of $E$ by $\Psi$ is the operation of replacing each occurrence of the variable in $E$ by an occurrence of the term in $T$, and the result obtained is called the **instance** of $E$ by $\Psi$, and is denoted by $E\Psi$.

### 2.2.2 Sequent Calculus

In 1933 Gentzen proposed the sequent calculus/natural deduction, a logical system that works on inference rules. Wang [Wan 60] produced programs which employ this proof procedure for quantificational theory.

If $\alpha$ and $\beta$ are wffs, then

- $\neg\alpha$
- $\alpha \lor \beta$
- $\alpha \land \beta$
- $\alpha \supset \beta$
- $\alpha \equiv \beta$
- $(Ex)\alpha$ and
- $(x)\alpha$

are wffs, where $x$ is a variable and the connective symbols: $\lor$, $\land$, $\supset$, $\equiv$, $(Ex)$ and $(x)$ are logical or, logical and, implication, biconditional, "some" and "all", respectively.

A string may be empty or a single wff. If $\mathcal{S}$ and $\Delta$ none-empty strings, then:

$\mathcal{S}, \Delta$

is also a string. Given two strings $\mathcal{S}$ and $\Delta$:

$\mathcal{S} \to \Delta$
is a sequent which intuitively speaking, is true if and only if either some wff in the string $\mathcal{J}$ (the 'antecedent') is false or some wff in the string $\Delta$ (the 'consequence') is true, i.e. the conjunction of all wffs in the antecedent implies the disjunction of all wffs in the consequence.

There are eleven rules of derivation. An initial rule states that a sequent with only atomic formulae is a theorem if and only if the same wff occurs on both sides of the arrow. There are two inference rules for each of the five truth functions - one introducing it (a truth function) into the antecedent and the other introducing it into the consequence.

Rule1 (initial rule): If $\mathcal{J}$ and $\Delta$ are strings of atomic formulae, then

$$\mathcal{J} \rightarrow \Delta$$

is a theorem if some atomic formula occurs on both sides of the arrow.

Inference Rules:

Rule2:

(a) If $\mathcal{J}$, $\alpha$, $\beta \rightarrow \Delta$, then $\mathcal{J}$, $\alpha \land \beta \rightarrow \Delta$

(b) If $\mathcal{J} \rightarrow \Delta$, $\alpha$ and $\mathcal{J} \rightarrow \Delta$, $\beta$ then $\mathcal{J} \rightarrow \Delta$, $\alpha \land \beta$

Rule3:

(a) If $\mathcal{J}$, $\alpha \rightarrow \Delta$ and $\mathcal{J}$, $\beta \rightarrow \Delta$ then $\mathcal{J}$, $\alpha \lor \beta \rightarrow \Delta$

(b) If $\mathcal{J} \rightarrow \Delta$, $\alpha$, $\beta$, then $\mathcal{J} \rightarrow \Delta$, $\alpha \lor \beta$

Rule4:

(a) If $\mathcal{J} \rightarrow \Delta$, $\alpha$, then $\mathcal{J}$, $\neg \alpha \rightarrow \Delta$

(b) If $\mathcal{J}$, $\alpha \rightarrow \Delta$, then $\mathcal{J} \rightarrow \Delta$, $\neg \alpha$

Rule5:
(a) If \( \mathcal{S} \rightarrow \Delta, \alpha \) and \( \mathcal{S}, \beta \rightarrow \Delta \), then \( \mathcal{S}, \alpha \supset \beta \rightarrow \Delta \)

(b) If \( \mathcal{S}, \alpha \rightarrow \Delta, \beta \), then \( \mathcal{S} \rightarrow \Delta, \alpha \supset \beta \)

Rule 6:

(a) If \( \mathcal{S}, \alpha \rightarrow \Delta, \beta \) and \( \mathcal{S}, \beta \rightarrow \Delta, \alpha \), then \( \mathcal{S} \rightarrow \Delta, \alpha \equiv \beta \)

(b) If \( \mathcal{S}, \alpha, \beta \rightarrow \Delta \) and \( \mathcal{S} \rightarrow \Delta, \alpha, \beta \),

then \( \mathcal{S}, \alpha \equiv \beta \rightarrow \Delta, \mathcal{S} \)

Given a sequent, we can find the first logical connective, i.e. the leftmost connective in the whole sequent and apply the appropriate rule to eliminate it. This results in one or two premises which taken together are equivalent to the conclusion. This process can be repeated until we reach a finite set of sequents with atomic formulae only. Each connective-free sequent can then be tested to see whether or not it is a theorem by using the initial rule. If all of them are theorems, then the original sequent is a theorem and a proof is obtained.

Let us demonstrate the use of the sequent calculus in a proof of the absorption laws:

\[ \alpha \land (\alpha \lor \beta) \equiv \alpha. \]

The sequent from the law is:

\[ \rightarrow \alpha \land (\alpha \lor \beta) \equiv \alpha. \]

To eliminate '≡', we apply rule 6 (a) (backwards) to get:

(a) \( \alpha \land (\alpha \lor \beta) \rightarrow \alpha \) and

(b) \( \alpha \rightarrow \alpha \land (\alpha \lor \beta) \)

First, we consider (a):

Applying rule 2(a) to (a) (to eliminate \( \land \)), gives:

\( (a') \alpha \), \( (\alpha \lor \beta) \rightarrow \alpha. \)
Finally, applying rule3(a) to \( (a') \) (to eliminate \( V \)), we have:

\[
(a'') \alpha, \alpha \rightarrow \alpha \text{ and } \alpha, \beta \rightarrow \alpha.
\]

Second, let us consider (b):

By applying rule2(b) to (b) (to eliminate \( \Lambda \)), we obtain:

\[
(b') \alpha \rightarrow \alpha \text{ and } \alpha \rightarrow (\alpha \lor \beta)
\]

Lastly, applying rule3(b) to (b') (to eliminate \( V \)) produces:

\[
(b'') \alpha \rightarrow \alpha \text{ and } \alpha \rightarrow \alpha, \beta.
\]

We observe that (by initial rule) the sequents in \( (a'') \) and \( (b'') \) are all theorems, therefore, the absorption law is a theorem and it is proved.

(The interested reader can find more material on sequent calculus in [Wan 60]).

2.2.3 A Refutational system

Wang's program, because of its use of Gentzen-like methods in quantificational theory, involves exponentiation on the truth functional connectives. Hence, later in the same year, David and Putnam [DaPu 60] gave a uniform proof procedure for quantificational theory which is feasible for use with some rather complicated wffs and which does not ordinarily lead to exponentiation. They invented a refutational system which proves theorems by contradiction. We consider their inference rules and give an illustration of how their system works. This will be compared with the system proposed by Robinson [Rob 63] which will be described later.
Inference rules:

1. Unit clause rule:
   \[ R \land (L \land \neg LVC) = R \lor \neg C \]
   i.e. if a literal \( L \) appears as a clause in a sentence \( S \), then \( S \) can be modified by striking out the clauses containing \( L \) and deleting every \( \neg L \) in other clauses. The resulting sentence \( S' \) is satisfiable if and only if \( S \) is satisfiable. A special case is given by the rule:
   \[ L \land \neg L = \text{NIL (contradiction)} \]
   i.e. if a sentence contains clauses of single literals \( L \) and \( \neg L \), then, it signifies that \( S \) is unsatisfiable.

2. Purity rule:
   \[ R \land (LVC) = R \quad \text{if } \neg L \text{ is not contained in } R \]
   i.e. if a literal \( L \) occurs in a sentence \( S \), only positively (or negatively), then all clauses containing \( L \) may be deleted. The remainder, \( S' \) is satisfiable if and only if \( S \) is satisfiable.

3. Split rule:
   \[ S = S_1 \lor S_2 \]
   i.e. if a sentence \( S \) contains a literal \( L \) in some clauses and \( \neg L \) in some others, \( S_1 \) may be obtained by striking out the clauses containing \( L \) and deleting the \( \neg L \)s from the others; similarly \( S_2 \) may be obtained by striking out the clauses containing \( \neg L \) and deleting the \( L \)s from the others. \( S \) is unsatisfiable, if and only if \( S_1 \) and \( S_2 \) are both unsatisfiable.
An example (from [Jou 88]) to illustrate the methodology is given below. If a sentence \( S \) is given as:

\[
S = p \lor q \land \neg q \lor s \land \neg s \lor p \land \neg p \lor r \land \neg p \lor r \lor t \land \neg r \lor t
\]

by:

(a) **split rule**: if \( p = \text{true} \) then, if \( p = \text{false} \) then,

\[
S_1 = \neg q \lor s \land r \land \neg r \lor t \land \neg t
\]

\[
S_2 = q \land \neg q \lor s \land \neg s \land \neg r \lor t
\]

(b) **unit clause rule** (on \( r \))

notice that in (a) we have \( r \) as a clause and \( \neg r \) in some other clauses, so uniting them we get:

\[
S_1 = \neg q \lor s \land t \land \neg t
\]

\[
S_2 = s \land \neg s \land \neg r \lor t
\]

(c) **unit clause rule** (on \( t \))

now the special case is satisfied with the literals \( t \) and \( \neg t \) which reduces to:

NIL (contradiction)

\( S_1 \) is unsatisfiable,

Therefore \( S \) is unsatisfiable.

**unit clause rule** (on \( q \)):

here uniting the first single literal clause which is \( q \), we have:

\[
S_2 = s \land \neg s \land \neg r \lor t
\]

\( S_2 \) is unsatisfiable.

**unit clause rule** (on \( s \)):

similarly, the special case is satisfied by considering the literal \( s \), given:

NIL (contradiction)

\( S_2 \) is unsatisfiable.
2.2.4 Resolution Systems

(A) Robinson [Rob 63]

Robinson proposed the ground resolution of predicate calculus as an alternative proof system. This involves computing the resolvents of pairs of clauses in a sentence in an attempt to establish if the sentence is satisfiable or unsatisfiable. The system was machine oriented and hence it was practical to use only one complex inference rule. This is in direct contrast to the human oriented system proposed by David and Putnam. However, during the computation of resolvents a lot of searching was involved. The ground resolution theorem [Rob 63] states that a sentence $S$ is unsatisfiable if and only if the (ground) resolution of $S$ contains NIL. An illustration of the use of ground resolution to show the unsatisfiability of the previously considered sentence:

$$S = p \lor q \land \neg q \lor s \land \neg s \lor p \land \neg p \lor r \land \neg p \lor s \lor r \lor t \land \neg r \lor t$$

is given in figure 2.2 (from [Jou 88]).
In the figure, the gaps between the clauses of S represent the 'A' operators.

(b) **Robinson** [Rob 65]

The consideration of predicates that contain variable arguments brought an immediate problem i.e. a predicate P(x), where x is a variable, indicates that x can take an indefinite (possibly infinite) number of values. In an attempt to find the proper finite set of instances from the infinite domain, Robinson developed a methodology called 'unification'.

**Definition**: A **unification** is a process whereby a (non-empty) set of wffs \( F \) and a substitution \( \Psi \) produce a (most general)
common instance \( \Psi \), and \( \Psi \) is called the most general unifier (mgu).

For example, given the predicates:

\[ P_1 = p(x, \Phi(x), \Psi(\Phi(w))), \text{ and} \]

\[ P_2 = p(\Psi(y), z, \Phi(y)) \]

where \( x, y \) and \( z \) are variables. We now demonstrate how a most general instance of \( P_1 \) and \( P_2 \) is obtained. First compare the predicate symbols, if these are identical, then unify arguments in sequence. Since the predicate symbols are the same we then continue as follows:

(a) if \( x \) becomes \( \Psi(y) \), then \( P_1 \) is transformed to:

\[ P(\Psi(y), \Phi(\Psi(y)), \Phi(\Phi(w))). \]

(b) if \( z \) becomes \( \Phi(\Psi(y)) \) (in (a)), then \( P_2 \) transforms to:

\[ P(\Psi(y), \Phi(\Psi(y)), \Phi(y)). \]

(c) if \( y \) becomes \( \Phi(w) \), then:

\[ P_1 = P_2 = p(\Phi(\Phi(w)), \Phi(\Phi(\Phi(w))), \Phi(\Phi(w))) \]

is the most general instance.

Therefore, the most general unifier (mgu) is the substitution:

\[ x \rightarrow \Phi(\Phi(w)) \]
\[ z \rightarrow \Phi(\Phi(\Phi(w))) \]
\[ y \rightarrow \Phi(w). \]

Robinson gave the algorithm for finding the mgu and showed that resolution is 'complete' for the first order predicate calculus.
A theorem proving system consists of a set of inference rules and a search plan. Therefore, a practical theorem proving strategy is required to be:

- **sound** - anything deduced from the inference rules is the logical consequence,
- **complete** - the strategy can derive everything that logically follows from the input, and
- **efficient** - the strategy should be able to prove theorems under reasonable time and space constraints.

### 2.3 Term-rewriting Methods

Many years after the discovery of resolution, in the search for complete and efficient strategies, term-rewriting methods were devised. These strategies use directed equations instead of undirected ones. We illustrate the difference between the use of ordinary and directed equations in the following proofs:

1. **Equational proof**: let the set of equations be:

   (a) \( x+0 = x \)
   (b) \( x+S(y) = S(x+y) \)

   using these equations we are to show that: \( S(x)+0 = x+S(0) \).

   **proof**: we start with the left hand side (lhs) and try to obtain the right hand side (rhs).
\[ S(x) + 0 = S(x) \quad \text{by (a)} \]
\[ = S(x + 0) \quad \text{by (a)} \quad \text{(in the opposite direction)} \]
\[ = x + S(0) \quad \text{by (b)} \]

2. Rewrite proof: the first obvious difference is that the equations are now directed to give:

(a) \( x + 0 \rightarrow x \)

(b) \( x + S(y) \rightarrow S(x + y) \).

(No specific methodology is utilized at the moment in orienting these equations. However, orientation is not at all at random, but we leave the discussion for the next chapter). Continuing with our proof, we are to show that:

\[ S(x) + 0 = x + S(0) \]

**proof:** unlike before, both sides are rewritten and the results are compared.

(i) \( S(x) + 0 \rightarrow S(x) \quad \text{by (a)} \)

(ii) \( x + S(0) \rightarrow S(x + 0) \quad \text{by (b)} \)
\[ \rightarrow S(x) \quad \text{by (a)} \]

Since both rewrite to the same term: \( S(x) \), we conclude that:

\[ S(x) + 0 = x + S(0) \]
2.3.1 Definitions and Notations

Let '$\rightarrow$' be a binary relation. Then $\rightarrow$ is **transitive** if:

\[ t \rightarrow s \text{ and } s \rightarrow u \Rightarrow t \rightarrow u. \]

and $\rightarrow$ is **irreflexive** if

\[ t \nrightarrow t \]

is not allowed.

Let '$\rightarrow^*$' denote closure of $\rightarrow$. A TRS is **terminating** if there is no infinite sequence of rewrites:

\[ t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow \ldots \]

A TRS is **locally confluent** if for terms $t, s$ and $u$, $s$ rewrites to $t$ and $s$ rewrites to $u$ (in just one step) implies there exists some term $v$ such that (in a finite number of steps), $t$ rewrites to $v$ and $u$ rewrites to $v$. This can be best represented diagrammatically as:

![Diagram](image)

A TRS is **confluent** if:

![Diagram](image)

A term-rewriting system is **complete** if it is both confluent and terminating.

A **position** (or an **occurrence**) is an identification number for a non-ground subterm of a term, and it is denoted by $o$. 
The subterm of a term $t$ at an occurrence $o$ is denoted by $t/o$. For example, in the term (given in a tree form):

```
x
/\ *
/  \
1   2
```

the numbers 1 and 2 indicate the left and right paths of each subtree. The occurrence 2 associates with the subterm $y*z$. Similarly, the number 21 indicates the leaf $y$. If $y$ is ground, i.e. a constant, then it is not an occurrence.

Let $\ell \to r$ and $g \to d$ be two rewrite rules in a TRS, $o$ an occurrence in $\ell$ such that $\Psi$ is a most general unifier of the subterm of $\ell$ at $o$ and $g$. Then $(r\Psi, \ell\Psi[d\Psi]_o)$ is a **critical pair** of $g \to d$ on $\ell \to r$ at $o$.

Let us illustrate a critical pair from the two rules $\ell \to r$ and $g \to d$ given as $x\lor(y\land z) \to (x\lor y) \land (x\lor z)$ and $x\land \neg x \to \text{false}$ respectively. Further let an occurrence in $\ell$ be 2, and $\Psi$ be a substitution given as follows:

$\Psi : y \to x$

$z \to \neg x$.

Then $\ell = x\lor(y\land z)$, $g = x\land \neg x$ and $\ell/2 = y\land z$. Unifying the subterm and $g$ using the substitution $\Psi$, we obtain the critical term $\ell\Psi$ (or $\ell\Psi[g\Psi]_2$):

$x\lor(x\land \neg x)$

This can be rewritten using both rules to give:

$x\lor x\land x\lor \neg x$  

($=r\Psi$)  

by rule $\ell \to r$

and
Knuth and Bendix gave the algorithm for generating a complete rewriting system, in particular for group theory. The process is based on the computation of critical pairs from a given set of directed axioms [KnBe 70]. When left members of the directed axioms overlap and the critical pair has distinct irreducible forms (structures), then a new directed axiom (consisting of the pair) must be added. The procedure is applied recursively until it eventually stops with either a success, i.e. a complete set is obtained, or a failure, i.e. a critical pair is produced which cannot be oriented. At times the procedure may not terminate, i.e. a new critical pair is always produced and oriented and process goes on and on.

2.3.2 Orderings

The proof of termination is based on 'well-founded' orderings.

**Definition:** A partial ordering $>$ is a transitive and irreflexive binary relation. It is total if, for any two terms $s$ and $t$, either $s > t$ or $t > s$.

**Definition:** The ordering $>$, is well-founded if it has no infinite descending sequence such as:

$$t_1 > t_2 > ... > t_n > ...$$
**Definition:** An ordering \( > \) is said to have the replacement property if \( s > t \) implies

\[
\mathcal{F}(u_1, \ldots, u_{i-1}, s, u_{i+1}, \ldots u_n) > \mathcal{F}(u_1, \ldots, u_{i-1}, t, u_{i+1}, \ldots u_n)
\]

for any term \( \mathcal{F}(u_1, u_2, \ldots u_n) \), \( s \) and \( t \). Note that the replacement must be at the same occurrence in the term.

The idea of the proof of termination, is to find a well-founded ordering \( > \), on the terms, such that \( s \) rewrites to \( t \) implies that \( s > t \). An ordering \( > \) which achieves this, is required to have the following properties:

- replacement;
- closure under instantiation, that is, \( s > t \) implies \( s \phi > t \phi \) for some substitution \( \phi \);
- subterm, that is, \( \mathcal{F}(x, y) > x \);
- for every rule \( \ell \rightarrow r \), then \( \ell > r \).

**Definition:** An ordering which has both the replacement and subterm properties is called a simplification ordering.

Dershowitz shows that any simplification ordering is well-founded [Der 82].

**Polynomial interpretation** is one of the many ways of demonstrating termination of some term-rewriting systems. It works by associating an integer polynomial of degree \( n \) with each \( n \)-ary operator, then composing these polynomials to interpret terms and comparing interpretations as positive functions ([Lan 75] and [Lan 79], also see [Der 85]). We illustrate a polynomial interpretation on the system:
\[ x + 0 \rightarrow x \]
\[ x + S(y) \rightarrow S(x + y) \]

where S and 0 are the successor and constant operator symbols respectively. If the constant operator symbol 0 is associated with the constant polynomial 1, S with the polynomial \( x + 2 \) and + with the polynomial \( (x + y)^2 \); then,

1. \((x + 0)\) (lhs of the first rule), is interpreted by \((x + 1)^2\) and \((x + 1)^2 > x\) for every \(x\)

2. \((x + S(y))\) (lhs of the second rule) is interpreted by \((x + y + 2)^2\) and \(S(x + y)\) (rhs of the second rule) by \(((x + y)^2 + 2)\), and it is obvious that \((x + y + 2)^2 > ((x + y)^2 + 2)\) for every \(x\) and \(y\).

Thus, the above system terminates with respect to the polynomial interpretation for all values of \(x\) and \(y\).

2.3.2.1 Orderings Based on Operator Precedence

The recursive path ordering (rpo) of Dershowitz, the path of subterm ordering (psd) of Plaisted and recursive decomposition ordering of Jouannaud et al are examples of simplification orderings ([Der 82], [Pla 78B], [JLR 82]). They are based on a (algebraic) precedence ordering on the operators.

A multiset (or bag) is an unordered collection of elements in which multiple occurrences are allowed. A partial ordering on any given set \(S\) can be extended to form an ordering on finite multisets over \(S\). In this extended ordering, a multiset \(M\) of \(S\) is considered 'smaller' than another multiset \(M'\) of \(S\) if \(M\) is obtained from \(M'\) by replacing one or more
elements in $M'$ by any (finite) number of elements taken from $S$, each of which is smaller than one of the replaced elements. The multiset ordering was proposed by Dershowitz and Manna ([DeMa 79], also see [JoLe 82]) and is the basis of many orderings used for proving termination of TRSs.

The idea of pso is that a term is decreased by replacing a subterm with any number of smaller (recursively) subterms connected by any structure of operators smaller, (in the total operator ordering), than the outermost operator of the replaced subterm.

**Definition:** Let a partial ordering $>$ be given on a set of operator symbols. Then, the recursive path ordering $>_\text{rpo}$, on the set of terms over the operator symbols is defined as follows:

$$t=\mathcal{F}(t_1,\ldots,t_n) >_\text{rpo} \mathcal{G}(s_1,\ldots,s_m)=s$$

iff

- $\mathcal{F}>\mathcal{G}$ and $(t) \gg \{s_1,\ldots,s_m\}$ or
- $\mathcal{F}=\mathcal{G}$ and $(t_1,\ldots,t_n) \gg \{s_1,\ldots,s_m\}$ or
- $\mathcal{F}<\mathcal{G}$ and $(t_1,\ldots,t_n) \geq \{s\}$

Here '>>' is the extension of $>_\text{rpo}$ to multisets and $\geq$ means $\gg$ or $=$. Dershowitz, also shows that, rpo is a 'complete' simplification ordering if the precedence ordering on the set of operator symbols is total [Der 82].

However, most recently, Jouannaud has highlighted the incompleteness of rpo and similar orderings, because, there
are some rewriting systems which terminate, but cannot be oriented in the same direction by rpo [Jou 88]. In particular, the rewriting system consisting of one rewrite rule:

\[ aba \rightarrow aa, \]

which terminates using an ordering based on the length of string, must be oriented to the opposite direction using rpo (assuming that \( a \) has higher precedence than \( b \)).

2.3.3 Term-rewriting Modulo Equations

Knuth and Bendix's completion algorithm failed because some equations could not be oriented to produce rewrite rules. The trend in generalizing the Knuth-Bendix procedure was to establish procedures that can handle the case when some equations destroy the finite termination property of the reduction relation. The general approach pursued for resolving this difficulty was to divide the set of axioms into two groups: the set of rules \( \mathcal{R} \) (for reduction relations) and the set equations \( \mathcal{E} \), for generating a congruence relation on the set of terms. The problem then is to develop algorithms that essentially operate on the congruence classes of terms rather than on the set of terms.

**Definition:** A quasi ordering \( \geq \) is a transitive and reflexive binary relation.

**Definition:** A TRS is quasi terminating if all (infinite) transformations contain only a finite number of different terms.
A quasi-reduction ordering $\geq$ whose associated equivalence relation contains the equational relation $\equiv_E$ needed to be developed in order to handle a term-rewriting modulo equations. Then a term $s$ quasi-reduces to a term $t$, if there exists a substitution $\phi$ and a rule $l \rightarrow r$ in $\mathcal{R}$, such that $l\phi = u$ for some subterm $u$ of a term $v$ where $v \equiv_E s$ in the equational theory $E$; and $t$ is equal (in $E$) to $v$ with $u$ replaced by $r\phi$. Some of the studies in this area are in: [Hue 80], [PeSt 81], [JoKi 83] and [Buc 87].

2.3.4 Completion

Completion, roughly speaking, is a procedure for obtaining a confluent and terminating TRS. The confluence test on TRS is based on two key theorems:

Theorem 2.1 (by Newman [New 42]): If TRS is terminating then confluence is equivalent to local confluence.

Theorem 2.2 ([KnBe 70] and [Hue 80]) A TRS is locally confluent if and only if all critical pairs are confluent.
**Ordinary Completion** - the completion of a term-rewriting system containing only rules. The procedure involves:

1. computation of non-confluent critical pair
2. orientation of the critical pair
3. normalization of new rule by TRS
4. addition of the rule to TRS
5. go to 1 and repeat until no further critical pair is produced.

**Unfailing Completion**

In an unfailing system (extended term-rewriting), there is no distinction between the rules and the equations. Let \( \lambda = r \) be an equation and \( \phi \) a substitution, if \( \lambda \phi > r \phi \), then \( \lambda \phi \rightarrow r \phi \) is the appropriate rewrite otherwise \( r \phi \rightarrow \lambda \phi \) is the rewrite. Since equations may be used in both directions, the extended critical pairs must take care of the two orientations. The unfailing completion procedure involves:

- computation of critical pairs
- simplification of equations
- subsumption of equations, that is, if \( s = t \) is obtained and there exists an equation \( \lambda = r \) in the set, such that, \( s = \lambda \phi \) and \( t = r \phi \) for some substitution \( \phi \), then \( s = t \) is assumed to be contained in the set.

The unfailing completion ensures confluence on ground terms only.

**E-Completion**

An E-term-rewriting system consists of a set of rules \( \mathcal{R} \) and a set of equation \( E \). The employed quasi-simplification
ordering is such that the associated congruence contains equality, and $l$ is greater than $r$ for each rule $l \rightarrow r$ in $R$. The E-critical pairs are computed between rules and between rules and equations.

Completion in Order Sorted Algebra
Equational deductions involve replacement of equals by equals and context application of lemmas. The completion needs the assumption that every rule $l \rightarrow r$, for all $\Phi$, $\text{leastsort}(l\Phi) \geq \text{leastsort}(r\Phi)$, where $\text{leastsort}(s)$, denotes the least sort to which $s$ belongs [SmNu 87].

2.3.5 Narrowing
Given a term-rewriting system, a substitution $\Phi$ and a term $t$, we say that $t$ narrows to $t'$ at an occurrence $o$ by rule $l_k \rightarrow r_k$, if and only if, $t/o$ is unifiable with $l_k$ and $\Phi$ is the most general unifier of $t/o$ and $l_k$ [DaGu 88]. Narrowing extends term-rewriting by using unification instead of matching to achieve rewriting. Narrowing was first used by Lankford [Lan 77].

2.4 Comparison of Simplification orderings
Steinbach [Ste 88] shows that by incorporating the principle of status on the operator, which determines the order in which the subterms are compared, some of the existing path and decomposition ordering can be extended. According to Forgaard [DeFo 85], the rpo with status differs from the ordinary rpo, only when it compares two terms whose top operator symbols are of equivalent precedence. In this case,
it refers to a part of the registry called the "status map", which maps each operator to a "status". This status information tells the ordering how to proceed in such a case. Three status map values are used in rpo with status. The multiset status, which indicates that the argument list of the terms being compared should be treated as multisets. The two lexicographic "statuses": left-to-right and right-to-left indicate that the arguments should be compared by forming sequences of terms by taking the arguments in forward and reverse order, respectively, and then lexicographically comparing these sequences [DeFo 85]. For more information on ordering with status (see also [Les 87]).

For any two orderings, either of the following holds:
- they are equivalent, or
- they overlap, or
- one is contained in the other.

The improved definition of the recursive decomposition ordering with status (rdos) by Rusinowitch, emerges to contain the rpo and pso (both re-defined with status). Also, it proves to be equivalent to the path ordering with status in [KNS 1985] and overlaps with the weight based ordering of Knuth and Bendix (also re-defined with status) (see, [KnBe 70], [Pla 78], [Rus 87] and [Ste 88]).

2.5 Conclusion
In the chapter we gave some events before the advent of the term-rewriting strategy and a brief discussion on the term-rewriting itself. The notion of consistency made it
impossible to ensure the completeness of a system. However, some completion procedures for some equational theories were discussed. Also included in the chapter was a comparison of some orderings. Steinbach [Ste 88], claimed that any two orderings may either be equivalent, or one contains the other or they overlap. Also, incorporating some status into an ordering, tends to extend it.
3. DERIVATION OF HIERARCHICAL OPERATOR ORDERING

Our TRS is based on a hierarchical operator ordering (HOO) which is derived from an appropriate operator precedence grammar. It is a total order relation defined over the set of all expressions definable in the language under consideration. HOO is not in general a simplification ordering; however, when used in the orientation of equations it qualifies as one. Before going into the formal derivation of the hierarchical operator ordering, we need to discuss some relevant grammatical pre-requisites. In the first section we shall give the grammatical origin of the ordering, and the derivation of a precedence relation from grammars. Non-ambiguous semantic trees from a grammar are called standard trees. Section 3.2 presents the hierarchical operator ordering formally. Also described is the use of the ordering in the orientation of equations.

3.1 Pre-requisites

Before defining HOO we now formalize its association with the related language. This is followed by the derivation of operator ordering from operator precedence grammars and the relationship between the semantic trees and standard trees

3.1.1 Grammatical Pre-requisites

Definition A letter (or character) is a single indivisible symbol and an alphabet is a set of letters.
**Definition** A *string* is an ordered collection of letters from an alphabet, A (repetition being allowed).

The possibility of no letters in the collection is also allowed and that gives an empty string which is denoted by $\lambda$. If A is set of symbols, then $A^*$ denotes the set of all finite strings from A including the empty string.

We will use the terms: sentence, expression and term interchangeably and to mean a string that is well-formed (i.e. a string that represents a wff).

**Definition** A *language* is a set of sentences.

A language L over an alphabet A is a subset of $A^*$, i.e. $L \subseteq A^*$. Most languages cannot be written out explicitly, in particular, the infinite ones, e.g. the language specifying the strings in which each string consists of an equal number of $a^*$, $b^*$, and $c^*$, in that order. The set of the sentences is:

\[
\{ \\
\text{abc,} \\
\text{aabbcc, (i.e. } a^2b^2c^2) \\
\text{a}^3b^3c^3, \\
\vdots \\
\text{a}^{100}b^{100}c^{100} \\
\text{\vdots} \\
\}
\]
As we can see there is an infinite number of these strings so it is not possible to list all of them but the set can be finitely represented as 

\[ L(G) = \{ a^n b^n c^n, \text{ and } n \geq 0 \}. \]

When \( n = 0 \) we obtain empty string. However, there are ways of generating the language; a grammar \( G \) may be referred to as such a generative system.

**Definition** [cf CoBe 84] A grammar \( G \) (context-free grammar or CFG) is an algebraic structure consisting of the ordered 4-tuple \( (N,T,P,S) \) where

(a) \( N \) and \( T \) are non-empty finite sets of non-terminal symbols and terminal symbols respectively, such that \( N \cap T = \emptyset \),

(b) \( P \) is a finite set of productions, \( P \subseteq N \times V^* \)

where \( V = N \cup T \) is called the vocabulary of \( G \), and

(c) \( S \in N \) and \( S \) is called the start symbol.

The production, \( (\alpha, \beta) \), is usually written as:

\[ \alpha \rightarrow \beta. \]

The symbol \( \rightarrow \) is not in \( V \). For example, \( G = (N,T,P,S) \) where:

\[ N = \{ S, A, B, C \}, \]
\[ T = \{ a, b, c \}, \text{ and} \]
\[ P = \{ S \rightarrow aA, A \rightarrow BbB, B \rightarrow aC, \]
\[ B \rightarrow bCc, B \rightarrow \Lambda, C \rightarrow cB, C \rightarrow \Lambda \}. \]

This grammar generates sentences some of which are:

\[ ab, abbc, aabc, aabc. \]

For convenience we are using upper case letters to denote the non-terminal symbols and the lower case letters for the
terminal symbols. Notice that in the set of productions we allow a non-terminal to transform to more than one string. By using the symbol: 'I' (meta-or), introduced by Backus in his definition of the syntax of Algol 60 [Bac 63], we can then represent the previous grammar as:

\[ G = (N, T, P, S) \]

where

- \( N = \{ S, A, B, C \} \),
- \( T = \{ a, b, c \} \), and
- \( P = \{ S \rightarrow aA, A \rightarrow BbB, B \rightarrow aC | bCc | \Lambda, C \rightarrow cB | \Lambda \} \).

Again, notice that we can deduce all the components of a grammar merely by listing the productions in a suitable order (one in which the start symbol appears on the lhs of the first production).

We now investigate how to get \( L(G) \) from a grammar \( G \). We say the string \( U \) is directly derived from \( W \), written

\[ W \Rightarrow U \]

if there exists a string say \( \alpha \) in \( W \) which was replaced by a string \( \beta \) by applying the production:

\[ \alpha \rightarrow \beta \]

to obtain \( U \).

Therefore, \( W \Rightarrow U \) implies that

\[ W = \delta \alpha \gamma, \]
\[ U = \delta \beta \gamma \]
and \( (\alpha \rightarrow \beta) \in P \), i.e. \( W \) is the juxtaposition of strings \( \delta, \alpha \) and \( \gamma \) etc. The language, \( L(G) \), generated by \( G \) is the set:

\[ \{ \gamma: \gamma \in T^* \text{ and } S \Rightarrow^* \gamma \} \]

(where \( \Rightarrow^* \) is read as 'derives').
The structure of an expression is best demonstrated using a tree. A tree showing the derivation of a sentence is called the derivation tree. Consider the sentence:

```
accbbcc
```

we show its derivation tree, using the grammar given earlier, in figure 3.1.

![Derivation Tree](image)

Figure 3.1

Reading the leaves from left to the right gives the sentence. The reverse process of what we have just done is known as the parsing of an expression, i.e. given an expression, we attempt to work our way backwards until we get to the root, the start symbol.

We are basically concerned with the structures of the sentences and not necessarily their meanings. However, there are some embedded meanings which we unconsciously take on board when considering the structures. Grammatically, all
terminal symbols are regarded as operator symbols, nevertheless, we should not confuse this with the semantical terminal symbols which refers to the non-operator symbols.

Consider the grammar with productions:

\[
\begin{align*}
E &\rightarrow T+E|T \\
T &\rightarrow T*P|P \\
P &\rightarrow (E)|a|b|c.
\end{align*}
\]

The expression, \(a+b*c\), using the productions, has the parse tree given in figure 3.2.

![Figure 3.2](image-url)
Semantically, figure 3.2 means that E is equal to the arithmetic structure in figure 3.3.

\[
+ \\
\text{a} \quad \text{b} \quad \text{c}
\]

Figure 3.3

The symbols + and * in figure 3.3 are not the leaves of the tree. This is one important requirements at the semantic level. These are leaves in figure 3.2 but are used to indicate the application of a computational process, and hence are represented differently in these two types of trees. This difference is showed diagrammatically in (a) and (b) of figure 3.4.

\[
\text{E} \\
+ \\
\text{a} \quad \text{b} \\
\text{f} \quad \text{g}
\]

(a) \hspace{1cm} (b)

Figure 3.4

It is the correspondence between these two types of trees which is essential to our methodology.
In the expression, \( a+b+c \), using the productions:

\[
E \rightarrow E + E \mid T, \quad \text{and} \\
T \rightarrow a \mid b \mid c
\]

we obtain two structurally different expressions:

\( (a+b)+c \) and \( a+(b+c) \).

The difference in the structure arises from the different parses which can be derived from this ambiguous grammar. In particular, the derivation given as follows:

\[
E \Rightarrow E + E \\
\Rightarrow E + E + T \\
\Rightarrow T + T + T \\
\Rightarrow a + b + c
\]

implies the ordering \((a+b)+c\), and the derivation,

\[
E \Rightarrow E + E \\
\Rightarrow T + E + E \\
\Rightarrow T + T + T \\
\Rightarrow a + b + c
\]

indicates the ordering \(a+(b+c)\).

However if we restrict the productions to left-recursive (LR) (or right-recursive (RR)) only one structure is possible, i.e.

\( (a+b)+c \) (or \( a+(b+c) \)).

If the operation denoted by the symbol + is associative, the semantics must indicate that:

\( (a+b)+c = a+(b+c) \),

and hence both evaluate to the same result.
The grammar that produces the expression \((a+b)+c\) is then given below:

\[
\begin{align*}
E & \rightarrow E+T \mid T, \text{ and} \\
T & \rightarrow a \mid b \mid c
\end{align*}
\]

The order in which the non-terminals must be reduced and consequently an order on the operators is generally known as precedence. This similarity of structure between the semantic and syntactic trees can be encapsulated in the formal definition of so called **Operator precedence grammars**.

**Definition**  Operator grammars are those context-free grammars in which all productions are such that no two non-terminal symbols are adjacent in the any right hand side and hence the intervening symbol may be thought of as an operator symbol (in the classical computational sense).

There are three precedence relations:

\(=\), \(\langle\) and \(\rangle\)

meaning 'equal precedence', 'lower precedence' and 'higher precedence' respectively. Their formal definitions (from [CoBe 84]) are as follows:

(a) \(a=b\) if \(A \rightarrow \alpha \beta b \gamma \in P\)

where \(\alpha, \gamma \in V^*\) and \(\beta \in \mathbb{N}U\{\Lambda}\)

(b) \(a\prec b\) if \(A \rightarrow \alpha A B \beta \in P\)

where \(B \Rightarrow ^+ \gamma \delta\), \(\alpha, \beta, \delta \in V^*\) and \(\gamma \in \mathbb{N}U\{\Lambda\}\), e.g. as in figure 3.4 where we have \(\prec\)

(b) \(a\succ b\) if \(A \rightarrow \alpha B b \beta \in P\)

where \(B \Rightarrow ^+ \gamma \alpha \delta\), \(\alpha, \beta, \gamma \in V^*\) and \(\delta \in \mathbb{N}U\{\Lambda}\).
Definition Operator precedence grammar is an operator grammar in which at most one precedence relation is defined between any two operator symbols.

3.1.2 Derivation of Operator Ordering
We now establish the formal link between operator precedence grammars and the hierarchical operator ordering we intend to present.

Recall the definition of a total ordering given in section 2.3.2 of chapter 2. We wish to define such an ordering on terminal symbols which we shall denote by \( \sigma \). For any two terminal symbols \( \mathcal{F} \) and \( \mathcal{G} \), \( \mathcal{F} \sigma \mathcal{G} \) implies that \( \mathcal{F} \) has a higher precedence than \( \mathcal{G} \).

Given the expressions:
1. \( a+b\cdot c \)
2. \( a\cdot b+c \)
3. \( a+b+c \)
where the operator symbols \( \cdot \) and \( + \) mean multiplication and addition respectively, the semantic trees for the expressions: 1 and 2 are in figure 3.5.

![Figure 3.5](image_url)
Notice that $+ < *$ in the first tree and $* > +$ in the second tree. Using the ordering, $\sigma$, we consider the paths in the trees, $\sigma$ will produce a complete ordering on the symbols contained. In the first tree we have the following:

1. $a \sigma +$,
2. $b \sigma * \sigma +$, and
3. $c \sigma * \sigma +$.

Similarly for the second tree. Another way of expressing the relation: $* > +$ is $+ < *$, but this we can not do using $\sigma$ because $\sigma$ is not symmetric.

The third expression:

$$a+b+c,$$

although it has two equal semantic trees (as mentioned earlier), considering the evaluation from left to right we will obtain the semantic tree:

```
       +
      /|
     / \\
   c   +
  / \
b a
```

indicating $+ > +$. Again, by our ordering we notice that:

1. $a \sigma + \sigma +$,
2. $b \sigma + \sigma +$, and
3. $c \sigma +$.

Indicating that we can have $\sigma \sigma \sigma$, where $\sigma$ is not a constant or a variable symbol. It is the reflexive property of $\sigma$ that allows this.
An operator precedence grammar may relate to an ordering on the operator symbols, it is this ordering we call $\sigma$ and we shall use it as a base to derive an ordering on the sentences which represent the expressions with which we wish to perform computations.

3.1.2.1 Standard Forms

An operator precedence grammar produces a parse tree from which the semantic tree is obtained. The semantic trees do not contain brackets and show no ambiguity, so we refer to them as the standard trees and a standard tree is said to be in its standard form (STF). Thus, a STF defines the same operator ordering as the semantic tree but extends the ordering to the constants and variables.

For instance, let the set of symbols of an algebra be given as:

$$T, G, \%, c_1, c_2, c_3, a, b, c$$

where $T$ and $G$ are binary operators, $\%$ a unary operator, $c_1$, $c_2$ and $c_3$ are constants and $a$, $b$ and $c$ are identifiers. A STF may express the ordering depicted in figure 3.6 on these symbols.
A grammar associated with the algebra is

\[
\begin{align*}
E & \rightarrow E \circ T \circ T, \\
T & \rightarrow T \circ P \circ P, \\
P & \rightarrow \circ Q \circ Q, \\
Q & \rightarrow c1 | c2 | c3 | a | b | c.
\end{align*}
\]

Figure 3.6 is satisfied by every path in any semantic tree obtained from the above grammar.

Within the sets of identifiers and constants, the lexicographic ordering (\(\sigma_{\text{lex}}\)) may be used to extend \(\sigma\). (When convenient we shall subsume \(\sigma_{\text{lex}}\) in \(\sigma\) and disregard the distinction). The extension of \(\sigma\) by the lexicographical ordering ensures that, at any time a complete ordering on the symbols (operators, constants and identifiers) is guaranteed (although, it is impossible to have two constants (or identifiers) along a single path in any tree).
A total ordering on the symbols defined by a STF is our basis for the definition of hierarchical operator ordering.

3.2 Presentation of Hierarchical Operator Ordering

We wish to formulate an order relation on the language $L(G)$ generated by the operator precedence grammar $G$. The elements of $L(G)$ are ground (meaning that they are composed of terminal symbols only), so that the use of lexicographical ordering can be justified. We shall call this order relation the 'Hierarchical Operator Ordering' (HOO) and denote it by $\rho$.

**Definition** Let a total ordering $\sigma$, on $T$, be given by the associated grammar $G$. We define a hierarchical operator ordering $\rho$, on $L(G)$ as follows:

for $t, s \in L(G)$ where $t \neq s$, $t = \mathcal{F}(t_1, \ldots, t_n)$ and $s = \mathcal{G}(s_1, \ldots, s_m)$, we say that $t \rho s$ if

1. $\mathcal{F} \rho \mathcal{G}$ and $m, n \geq 1$.

   or

2. $\mathcal{F} = \mathcal{G}$ and there is some $i$, $1 \leq i \leq m$ and $1 \leq i \leq n$ such that $t_j = s_j$ for $j < i$, $t_i \neq s_i$ and $t_i \rho s_i$.

   or

3. $n \geq 1$ and $m = 0$ (meaning that $s$ is either a constant or an identifier).

   or
4. $t$ and $s$ are both constants or identifiers and $t \preceq_{\text{lex}} s$
   
   (where $\preceq_{\text{lex}}$ denotes the lexicographic order with respect to $\preceq$)
   
   or
   
   $t$ is an identifier and $s$ is a constant.

To ensure the totality of $\preceq$ we also stipulate that $s \preceq s$ for all $s \in L(G)$. Note that $\preceq$ is not, in general, a simplification ordering as opposed to the path orderings and the decomposition ordering mentioned in chapter 2. However, they all share the same underlying operator precedence requirement. For any arbitrary terms $s$ and $t$, $t \preceq s$, simply means that $t$ is related to $s$ - i.e. that the structure of $s$ is in some sense preferable to that of $t$ (but not necessarily that $s$ is algebraically simpler than $t$). Furthermore, $\preceq$ is both transitive and irreflexive.

If $\preceq$ is required to serve as a simplification ordering, any two terms under consideration must be algebraically equal; only then would $t \preceq s$ imply that $s$ was (syntactically) "simpler" than $t$.

Let us demonstrate an application of $\preceq$, in the comparison of the two arbitrary terms:

\[ t = \mathcal{F}(\mathcal{F}(x, y), z) \quad \text{and} \quad s = \mathcal{F}(x, \mathcal{F}(y, z)) \]

\[ s = \mathcal{F} \quad \mathcal{F} \quad z \quad \mathcal{F} \quad \mathcal{F} \quad x \quad y \quad z \quad \text{and} \quad t = \mathcal{F} \quad \mathcal{F} \quad x \quad \mathcal{F} \quad y \quad z \]
We need to show how \( p \) applies to the terms. With reference to the definition of HOO, since the two outermost operator symbols in both terms are equal, it suffices to compare their immediate left arguments, which are:

\[
\begin{array}{c}
\& \\
x & \& y & \& \text{and} & \& x
\end{array}
\]

By the third option of the definition of \( p \):

\[
\begin{array}{c}
\& \\
x & \& y & \& \rho & \& x
\end{array}
\]

Since the arity of \( \& \) is 2 and that of \( x \) is zero. Therefore, we conclude that \( tps \).

3.3 Exploitation of Hierarchical Operator Ordering to Orientate Equations

Hierarchical operator ordering was defined in order to aid in the relating of terms. Any element of \( L(G) \) is either a variable or a constant or a combination of variables and/or constants and operator symbols. In that case, any two different terms will be either:

(a) both constants or
(b) both variables or
(c) constant and variable or
(d) constant and \( F(t_1, \ldots, t_n) \) (where \( F \) is an operator and \( t_1, \ldots, t_n \) are terms and \( n \geq 1 \)) or

(e) variable and \( F(t_1, \ldots, t_n) \) or

(f) \( F(t_1, \ldots, t_n) \) and \( G(s_1, \ldots, s_m) \) \( (m, n \geq 1) \)

In these six possibilities any pair can be \( \rho \)-related. Emphasizing the total order relation property of \( \rho \) on \( L(G) \).

3.3.1 Orientation of Equations

The main motive for developing \( \rho \) is for the orientation of equations to obtain rewrite rules. Equations are of the form:

\[ t = s \]

where \( t \) and \( s \) are terms. They show a symmetric relation, i.e. \( t = s \rightarrow s = t \). The \( \rho \) dislodges this symmetric property in the equations and gives them a sense of direction. An equation, say \( t = s \), orients towards \( s \) if and only if \( t \) is related to \( s \) - \( t \rho s \).

Equations indicate algebraically equal terms. When \( t = s \), it means that the identical variables in \( t \) and \( s \) represent the same values at any time. In other words, if a variable, say \( x \), in \( t \) takes the value \( a \) and there is also \( x \) in \( s \), it must be instantiated to \( a \). This may not be the case for arbitrary terms i.e. two unequal terms; since their variables are unbounded. For this reason we cannot say that one structure among the two is the simpler. But when considering equations, it does mean that one that has the preferred structure is simpler. Thus, in this situation, \( \rho \) serves as a
simplification ordering. The set \( L(G) \) constitutes a number of equivalent classes of expressions, therefore, for each class \( \rho \) is a simplification ordering. So, when expressions from different classes are considered \( \rho \) serves just as an "ordinary" order relation (with no reference to the simplicity of either terms).

Let us consider the orientation of some equations:

1. 
\[
\begin{array}{c}
\ast \\
\downarrow \\
-1 \\
\downarrow \\
x
\end{array}
\]
\[
x \ast -1 = 1
\]

2. 
\[
\begin{array}{c}
\ast \\
\downarrow \\
+ \\
\downarrow \\
\downarrow \\
y \\
z \\
\end{array}
\]
\[
x \ast + = y \ast z
\]

where the 'normal' order on the operators is: \( \ast \sigma^+ \). In the first equation, the operator symbol \( \ast \), on the lhs, is of arity 2 and the rhs is a constant, therefore, by the option 3 of the definition of the ordering, the equation orients to the right. Similarly, the second equation orients to the right because \( \ast \sigma^+ \), which satisfies the first option of the definition. The directed equations obtained form the above equations are thus:

1. 
\[
\begin{array}{c}
\ast \\
\downarrow \\
-1 \\
\downarrow \\
x
\end{array}
\]
\[
x \ast -1 \rightarrow 1
\]
2.

\[
\begin{array}{c}
\text{\(x\)} \\
\text{\(+\)} \\
\text{\(y\)} \\
\text{\(z\)}
\end{array} \quad \rightarrow \quad \begin{array}{c}
\text{\(x\)} \\
\text{\(+\)} \\
\text{\(y\)} \\
\text{\(x\)} \\
\text{\(z\)}
\end{array}
\]

3.4 Conclusion

We defined a new ordering \(\rho\), for orienting equations, and emphasized that \(\rho\) does not qualify as a simplification ordering when arbitrary terms are considered. It is a simple practical ordering that assists in the derivation of rewrite rules from equations to ensure the termination of a term-rewriting system based on these rules. The order relation \(\sigma\) on the symbols, from which \(\rho\) was derived, was the associated with the precedence of an operator precedence grammar, suitably extended to constants and identifiers.
4. A NEW STRATEGY FOR TERM-REWITING SYSTEMS I: AN EXAMPLE

As stated in chapter 2, confluence was an important criterion of the traditional term-rewriting system (TRS). Finite termination ensures that every term has a "final" form. Unique termination together with finite termination implies that every term has just one final form; and basically this is what the confluence property means. The reason why a term can possibly reduce to two syntactically different ones is because of the non-deterministic nature of general TRSs. In confluent systems rewriting can be done in any order. However, in a deterministic TRS there is no ambiguity whatsoever since one and only one order of rewriting is allowed. Thus, a test for the confluence property of such a system is irrelevant. One way of accomplishing determinism is by enforcing an order on rule applications.

For instance, the TRS consisting of the following set of rewrite rules:

\[
\{ \neg \neg x \rightarrow x, \\
\neg (x \lor y) \rightarrow \neg x \land \neg y, \\
\neg (x \land y) \rightarrow \neg x \lor \neg y, \\
x \land (y \lor z) \rightarrow (x \land y) \lor (x \land z), \\
(x \lor y) \land z \rightarrow (x \land z) \lor (y \land z) \}
\]

is not confluent.
This can be illustrated by considering the transformation of the term:

$$-(x \land (y \lor z)).$$

If the third rewrite rule is applied and then followed by the second rule the term yields

$$(-x \lor (-y \land -z)).$$

Alternatively, applying the fourth rule first followed by other applicable rules leads to the term:

$$(((\neg x \land \neg x) \lor (\neg x \land \neg z)) \lor ((\neg y \land \neg x) \lor (\neg y \land \neg z))).$$

Syntactically the two results are not equal and rewriting has terminated, and hence the non-confluence of the TRS is demonstrated. But if there is the restriction, such as 'first match first transform', then the term will yield only

$$(-x \lor (-y \land -z)).$$

So, there is no ambiguity. In a sequential and deterministic system like the one we will discuss later, the leftmost-outermost transformations are found by a depth-first search from left to right.
There are three situations that can arise in a matching or searching process:

- **disjoint matches:**

- **an ancestor match:**

- **an overlap match:**

The three cases can be comfortably handled through the depth-first search. In case three, if there is no rewrite rule that matches the composite (of \( \ell \) and \( g \)) \( \ell \) will match before \( g \), using depth-first strategy. Also, some restrictions needs to be imposed in order to avoid a serious problem which might arise if the rewrite rules overlap. Such a dangerous overlapping occurs when two rewrite rules match the same ground term. In order to avoid this happening, a restriction due to O'Donnell [O'D 77] is imposed.
It requires that two rewrite rules with overlapping lhs's have the same corresponding rhs's, e.g. the rules:

\[ F(x, 1) \rightarrow 1 \]
\[ F(0, x) \rightarrow 0 \]

are prohibited since they both match the term \( F(0, 1) \); which can thus reduce to either 1 or 0.

Our strategy is a development of a TRS based on an order of rule applications. It is beneficial to define a unique standard form whenever possible. Since, if it is defined, there are no two ways about it, either we reach the desired form or we do not.

In the first section we shall present our new strategy. This has a remarkable interesting advantage over the traditional methods which are non-deterministic. The following section discusses the use of the HOO, presented in the previous chapter, in the orientation of the axioms of the Boolean algebra. In the last section the strategy is illustrated by describing the development of a TRS for a Boolean algebra. In it, a unique intermediate form is achieved and consequently simplified.

4.1. A New Approach

We developed a term-rewriting strategy defined via a sequence of intermediate processes. Each is driven by a set of rewrite rules. The choice of the rules used in each stage is determined by the sub-form which relates to the goal at that stage. The sets of rules used at each individual stage are
not necessarily distinct. To ensure that the same sub-form is always obtained for the same term, there is a definite order in which the rewrite rules are applied in each stage and this order is determined by reference to the current form of the expression.

The standard form, although unique, could be very lengthy or even infeasible as the number of the variables increases. So, to avoid the situation, a second form is defined, and its derivation from the first prompts the application of some previously applied rule in the opposite direction. A novel feature of our strategy is the ability to apply some rewrite rules in the opposite directions at different stages within the overall TRS. This is unthinkable in traditional systems.

4.2. Illustration of the Strategy as Applied to a Boolean Algebra
The Boolean algebra involving the minimal set of operators which are: the constants true and false and the operators \( \wedge \) (and), \( \vee \) (or) and \( \neg \) (not) is considered. If one wants to consider other connectives, this is not difficult. Development of a system to convert a boolean expression (containing any other operator(s)) to its equivalent containing only the above mentioned operators is straightforward provided that the combination of operators used does not admit two, 'normal' representations of any expression.
The system consists of two main phases. These correspond to the two important forms sought-after by the system. The first, which we shall call 'Principal Disjunctive Normal Form' (PDNF), is the familiar form consisting of logical sums (\(\lor\)) of products (\(\land\)) of basic terms which are distinct variables or their complements (\(\neg\)). This is then factorised to give what we call the 'Reduced Disjunctive Normal Form' (RDNF). The PDNF has a very lengthy representation, especially when the number of the variables in a given expression is large. This problem prompted the need for a more manageable form 'RDNF' which is obtained by simplifying the PDNF. The uniqueness of the PDNF makes it more suitable than the RDNF, for resolving the equivalence problem. Actually, PDNF represents an equivalence class of the algebra, therefore, two expressions are equivalent if both transform to the same PDNF.

Our methodology requires a detailed definition of a standard form because that determines most of the manipulations of the system. As the PDNF is unique, we prefer to use its definition.
The operators $\land$ and $\lor$ are both associative and commutative, so, utilizing these, the syntax definition of PDNF in BNF (see [Bac 63]), is given as:

\[
\begin{align*}
\text{<PDNF>} &::= \text{<ListofV-exp> | False} \\
\text{<ListofV-exp>} &::= \lor (\text{<ListofV-exp>}) \text{<\land-exp>}
\end{align*}
\]

\[
\begin{align*}
\text{<ListofV-exp>} &::= \text{<\land-exp>, <ListofV-exp> | <\land-exp>, <\land-exp>}
\end{align*}
\]

\[
\begin{align*}
\text{<\land-exp>} &::= \land (\text{<Listofprimitives>}) \text{<Primitive>}
\end{align*}
\]

\[
\begin{align*}
\text{<Listofprimitives>} &::= \text{<Primitive>, <Listofprimitives> | <Primitive>, <Primitive>}
\end{align*}
\]

\[
\begin{align*}
\text{<Primitive>} &::= \neg \text{id} | \text{id}
\end{align*}
\]

\[
\begin{align*}
\text{id} &::= \text{not defined}
\end{align*}
\]

The given syntax is dependent on the defined ordering on the operator symbols (from the associated grammar). In other words, if a different order is required on the symbols, we should expect a change on the order the symbols is given in the definition.

The number of primitives in \text{<Listofprimitives>} corresponds to the number, $n$, of distinct identifiers in an expression intended for standardization. Also, if \text{<Listofprimitives>} is the list:

\[
(id_1, id_2, \ldots, id_n),
\]

then, $id_{i+1} \preceq id_i$, for $i = 1$ to $n-1$.

The length of \text{<Listof\land-exp>} is required to be less than or equal to $2^n$, where there are $n$ distinct identifiers in the expression.

If $\langle \text{Listof} \Lambda \text{-exp} \rangle$ is the list:

$$(\Lambda \text{-exp}_1, \Lambda \text{-exp}_2, \ldots, \Lambda \text{-exp}_n),$$

then,

$$\Lambda \text{-exp}_{j+1} \neq \Lambda \text{-exp}_j$$

for $j = 1$ to $n-1$, i.e. if

$$\Lambda \text{-exp}_j = (id_1, id_2, \ldots, id_n) \text{ and }$$

$$\Lambda \text{-exp}_{j+1} = (id'_1, id'_2, \ldots, id'_n)$$

then from left to right the first $i$, such that $id_i \neq id'_i$ and $id'_i \neq id_i$.

Each $\Lambda \text{-exp}_i$ is called a 'minterm', i.e. a conjunction which contains the exact distinct number of variables (identifiers) as in a given expression. For instance, the set of minterms constructed from the two variables, $x$ and $y$ is:

$$\{ \Lambda(x, y), \Lambda(x, \neg y), \Lambda(\neg x, y), \Lambda(\neg x, \neg y) \}$$

Notice the number of minterms in the set, there are 4 ($= 2^2$). In general, for $n$ variables the number of minterms is given by $2^n$. For any expression in PDNF, the number of minterms contained is less than or equal to $2^n$, where $n$ is the number of distinct variables in the expression. However, equality holds only for tautologies which in the second phase of the reduction identically yields the value true.

4.2.1 The Orientation of Equations

We take a sufficiently large set of equations, which include the axioms of the boolean algebra and some of their consequences. This guarantees the successful application of
our strategy. The formal requirements of these 'equations' will be discussed at length in a later chapter. The set is given below:

1. \( x \lor y = y \lor x \)
2. \( (x \lor y) \lor z = x \lor (y \lor z) \)
3. \( \text{false} \lor x = x \)
4. \( x \lor \neg x = \text{true} \)
5. \( x \land y = y \land x \)
6. \( (x \land y) \land z = x \land (y \land z) \)
7. \( \text{true} \land x = x \)
8. \( x \land \neg x = \text{false} \)
9. \( x \land (y \lor z) = (x \land y) \lor (x \land z) \)
10. \( x \lor (y \land z) = (x \lor y) \land (x \lor z) \)
11. \( \text{true} \lor x = \text{true} \)
12. \( \text{false} \land x = \text{false} \)
13. \( x \lor x = x \)
14. \( x \land x = x \)
15. \( x \land (x \lor y) = x \)
16. \( x \lor (x \land y) = x \)
17. \( \neg \neg x = x \)
18. \( \neg (x \land y) = \neg x \lor \neg y \)
19. \( \neg (x \lor y) = \neg x \land \neg y \)
20. \( \neg (\text{false}) = \text{true} \)
21. \( \neg (\text{true}) = \text{false} \)
22. \( x \land \neg y = \neg y \land x \)
The PDNF defines the operator hierarchy:

\[
(false, true) \sigma \text{ ids } \sigma \rightarrow \sigma \land \sigma \lor
\]

The two constants are, legitimately, considered as operators and are ordered accordingly. The lexicographical ordering enables us to extend the ordering \( \sigma \) within the constant symbols; note that in practice the constants (and identifiers or ids), appear at the same level in a tree. It may seem that we are violating the semantic rule by saying that constants are operators. However, this allows to have a total ordering on the symbols involved. This ordering, \( \sigma \), on the operator symbols, is used to define the HOO \( p \) which is subsequently used to orient the above equations to obtain the set of rewrite rules where \( x, y \) and \( z \) are (expression) variables:

1. \( x \lor y \rightarrow y \lor x \) 
2. \( (x \lor y) \lor z \rightarrow x \lor (y \lor z) \)
3. \( false \lor x \rightarrow x \)
4. \( x \lor \neg x \rightarrow true \)
5. \( x \land y \rightarrow y \land x \)
6. \( (x \land y) \land z \rightarrow x \land (y \land z) \)
7. \( true \land x \rightarrow x \)
8. \( x \land \neg x \rightarrow false \)
9. \( x \land (y \lor z) \rightarrow (x \land y) \lor (x \land z) \)
10. \( x \lor (y \land z) \rightarrow (x \lor y) \land (x \lor z) \)
11. \( true \lor x \rightarrow true \)
12. \( false \land x \rightarrow false \)
13. \( x \lor x \rightarrow x \)
14. \( x \wedge x \rightarrow x \)
15. \( x \wedge (x \vee y) \rightarrow x \)
16. \( x \vee (x \wedge y) \rightarrow x \)
17. \( \neg \neg x \rightarrow x \)
18. \( \neg (x \wedge y) \rightarrow \neg x \vee \neg y \)
19. \( \neg (x \vee y) \rightarrow \neg x \wedge \neg y \)
20. \( \neg (\text{false}) \rightarrow \text{true} \)
21. \( \neg (\text{true}) \rightarrow \text{false} \)
22. \( x \wedge \neg y \rightarrow \neg y \wedge x \)

if \( x \neq y \).

In particular, rule (1) is obtained by applying the second option of the definition of the HOO. The outermost operator symbols of the lhs and rhs of equation (1) are equal so, considering their arguments from the left and looking for the first different ones, we obtain \( x \) and \( y \). The \( x \), \( y \) and \( z \) used in these rules are (expression) variables and hence their lexicographical ordering is undecidable; because the (ground) terms to which the variables will later be instantiated are unknown. However, we could still apply lexicographical ordering inductively, i.e. by considering the full transformation of the variables to ground terms. In this case, these are the constants (true, false) or any identifiers. Actually, the stage in the system where the commutative rules, which call for lexicographic order, are applied involves manipulation of ground terms. Thus, the lexicographical ordering is decidable.
4.2.2 **The Normalization Procedure**

Within the two main phases mentioned above, the system contains other stages that produce important sub-forms. Each stage establishes an order of rule application. Despite the segmentation, there is an order defined by the tree representing an expression in which the individual rules are applied within a stage. In the sections 4.2.2.1 through 4.2.2.5 we trace the normalization of an expression through all its stages.

4.2.2.1 **The Initial Expression**

We start from the assumption that initially the expression is valid, i.e. it is well-formed, syntactically (and with respect to type constraints when several types are involved). A well formed expression, i.e. a sentence in $L(G)$, is parsed using $G$ to obtain its syntax tree and subsequently its semantic structure tree. To illustrate with the expression:

$$((\neg(a \land \neg b) \lor c) \land d \lor a) \land (a \land (b \lor c)),$$

we have the productions of the operator precedence grammar associated to the Boolean algebra be given as:

- $E \rightarrow E \lor T|T,$
- $T \rightarrow T \land P|P,$
- $P \rightarrow \neg Q|Q,$ and
- $Q \rightarrow (E)|\text{false}|\text{true}|a|b|c|d.$

The small set of identifiers, \{a,b,c,d\}, is just for illustrative purposes as there could be many more of them. The parse tree for the above boolean expression is given in figure 4.1.
In figure 4.1 we used the triangles to abbreviate the parsing of the sub-expressions directly below the triangles, the fully parsed symbols indicate how the complete parse tree can be obtained. Figure 4.2 shows a simpler representation of figure 4.1.
Figure 4.2

The list representation of this is:

\[ \land[\lor[\land[\lor[\neg[\land[\neg[\land[\neg[a,\neg[b]
\lor[c,\neg[d,\neg[a,\lor[b, c,\land[a,\lor[b, c]]]]]]]]]]]]]]]]]

4.2.2.2 Adjusting Operator Ordering

This is the first stage of the rewriting process. The expression is checked to see whether its operators obey the operator hierarchy. The transformations which concern the
re-ordering of the operators are performed when the operator hierarchy is not consistent with $\sigma$. Below is a table which indicates rules that are applicable to cause the desired re-ordering. It is constructed from the set of operator symbols. The table gives the possible orderings of pairs of computational operators, and the number of the rewrite rule that will invoke the necessary re-ordering. In the table:

<table>
<thead>
<tr>
<th>2nd</th>
<th>false</th>
<th>true</th>
<th>$\neg$</th>
<th>$\land$</th>
<th>$\lor$</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>true</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>$\neg$</td>
<td>20</td>
<td>21</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>$\land$</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>9</td>
</tr>
<tr>
<td>$\lor$</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
<td>ok</td>
</tr>
</tbody>
</table>

Table 4.1

Showing the equations needed for operator re-ordering

ok means that the order is correct and nil indicates that such situation should not occur legally. The 1st and 2nd means the upper and lower respectively, in the tree structures of expressions.
The set of rewrite rules required at this stage is given thus:

\[
\begin{align*}
9. \quad x \land (y \lor z) & \to (x \land y) \lor (x \land z) \\
17. \quad \neg \neg x & \to x \\
18. \quad \neg (x \land y) & \to \neg x \lor \neg y \\
19. \quad \neg (x \lor y) & \to \neg x \land \neg y
\end{align*}
\]

Other rewrite rules are not included because they are not necessary for now. The numbering of the rules does not in any way reflect the order the rules are applied; it is merely a reference to the original set of rules. The order that rules are applied is strictly on depth-first left to right search bases. This method has an interesting property, that the transformation of the top match may cause some lower one(s) to transform, this may not be the case otherwise. On termination of the sub-system, the tree structure of the expression produced from that in figure 4.2 is given in figure 4.3.
Figure 4.3
Also, the list representation is:

\[ V[V[V[\land[\neg[a], d],
    \land[a, b]]],
    \land[\land[\neg[a], d],
    \land[a, c]]] \]

\[ V[\land[b, d],
    \land[a, b]] \]

\[ \land[\land[b, d],
    \land[a, c]] \]

\[ V[\land[c, d],
    \land[a, b]] \]

\[ \land[\land[c, d],
    \land[a, c]] \]

\[ V[\land[a,
    \land[a, b]]],
    \land[a, \land[a, c]] \]

We trace some of the transformation performed on the expression in the figure 4.2. The root matches the lhs of the distributive law - rule 9 (the commutative property of \( \land \)). By performing all the necessary substitutions of the sub-expressions to the variables in the lhs and replacing it by the rhs of the rule, again, with the same substitutions for the variables, the new tree structure obtained is given in figure 4.4.
The structure showing the result of a transformation at the root of the expression in figure 4.2

Figure 4.4

In the rewrite rule: \((x \lor y) \land z \rightarrow (x \land z) \lor (y \land z)\), when applied, the variables: \(x\), \(y\) and \(z\), on the lhs are substituted by the appropriate subterms of figure 4.2 and the same substitutions are made on the variables on the rhs to obtain figure 4.4. So, as the lhs relates to the rhs of the rules, so does figure 4.2 relates to figure 4.4; thus indicating that substitutions on the variables preserve the ordering. The rewriting continued in this manner until the expression given in figure 4.3 was obtained.
4.2.2.3 Restructuring and Simplifying of Expression Using AC Properties of Operators

The operator symbols that have both associative and commutative properties are said to be AC operators. We are able to do restructuring because of the associativity and commutativity of the operators: \( \land \) and \( \lor \). However, for operators that do not possess these properties, it is important to note that the sort of restructuring described below may not be applicable and in that situation this stage is missed out. The consequence will be more STFs. These properties are exploited in converting the binary operators to multi-operand operators. For instance, the expression: \( \land(a, \land(d, f)) \) is flattened to \( \land(a, d, f) \), and similarly for the operator \( \lor \). The sub-system that ensures that this can be done is given by the set of rules:

\[
\begin{align*}
2. \quad & (x \lor y) \lor z \rightarrow x \lor (y \lor z) \\
6. \quad & (x \land y) \land z \rightarrow x \land (y \land z)
\end{align*}
\]

The transformation from left association to right association is performed and from the right association the flattened form is derived. The order of application of these rules is still on "first match first transformed" basis. The realization of the flattened form is via some list handling procedures. For example, to derive \( \land(a, d, f) \) from \( \land(a, \land(d, f)) \), the list: \([d, f]\) is appended to the singleton: \([a]\) with the retention of the outer \( \land \). The restructuring of expression in figure 4.3 gives:
\[ \vee [\wedge \neg [a], d, a, b], \\
\wedge [\neg [a], d, a, c], \\
\wedge [b, d, a, b], \\
\wedge [b, d, a, c], \\
\wedge [c, d, a, b], \\
\wedge [c, d, a, c], \\
\wedge [a, a, b], \\
\wedge [a, a, c], \\
\wedge [a, a, c] \]

or the tree form in figure 4.5.

![Tree diagram](https://example.com/tree.png)

**Figure 4.5**

It can be deduced that the figure has almost the desired PDNF structure. The simplification process achieves some other characteristics of the PDNF. These include a specific order, the combination of multiples and removal of trivial factors with respect to the AC operator symbols: \( \wedge \) and \( \vee \).
The rewrite rules needed for the simplification at the $\land$ level are:

(a) 5. $x \land y \rightarrow y \land x$ if $x \not= y$
22. $x \land \neg y \rightarrow \neg y \land x$ if $x \not= y$

(b) 14. $x \land x \rightarrow x$

(c) 8. $x \land \neg x \rightarrow false$
12. $false \land x \rightarrow false$

The order in which these rules are applied is as labeled above, ((a) through (c)). In particular, the commutative rules are considered first, which includes the rule: $x \land \neg y \rightarrow \neg y \land x$ if $y \not= x$. It is realized by the lexicographic order on the identifiers and the constants. The effect enables us to bring identical terms adjacent to each other where they can conveniently be combined by use of idempotent rules. If there is a contradicting pair (an identifier and its complement) in an $\land$-expression, the whole $\land$-expression reduces to $false$. For that reason, the rules are applied in the order they are labeled (a) through (c). But the order the rules in (c) are applied is again determined by the expression.
Continuing with our example, the expression obtained after the commutative rule is:

\[
\lor[\land[a, a, b], \\
\land[a, a, c] \\
\land[a, \neg[a], b, d], \\
\land[a, \neg[a], c, d] \\
\land[a, b, b, d], \\
\land[a, b, c, d], \\
\land[a, b, c, d], \\
\land[a, c, c, d], \\
\land[a, c, c, d] 
\]

The tree representation is in figure 4.6.

![Figure 4.6](image)

Applying idempotent rule to figure 4.6 produces:

\[
\lor[\land[a, b], \\
\land[a, c] \\
\land[a, \neg[a], b, d], \\
\land[a, \neg[a], c, d] \\
\land[a, b, b, d], \\
\land[a, b, c, d], \\
\land[a, b, c, d], \\
\land[a, c, c, d], \\
\land[a, c, c, d] 
\]

and the tree structure in figure 4.7.
This in turn reduces to the expression below and figure 4.8.

\[ \lor[\land[a, b], \\
   \land[a, c], \\
   \text{false}, \\
   \text{false}, \\
   \land[a, b, d], \\
   \land[a, b, c, d], \\
   \land[a, b, c, d], \\
   \land[a, c, d]. \\
] \]

The above process is repeated, but this time with the \( \lor \) symbol. The methodology for processing can be viewed as 'bottom-up'. Having completed the processing at the leaves
(with respect to \( \land \)), we now move to the next level up, i.e. the processing of the \( \land \)-expressions with respect to \( \lor \). The application of the commutative rule at this level is based on its result at the \( \land \) level. That is, \( x \lor y \rightarrow y \lor x \) if
\[
x = \land(a_1, a_2, \ldots, a_n),
y = \land(b_1, b_2, \ldots, b_n)
\]
and from left to right there exists \( i \), such that \( a_i \neq b_i \), \( a_j = b_j \) for \( j < i \) and \( a_i \preceq b_i \).

The rewrite rules for simplifying the \( \lor \) level and the order they are applied are given below:

(a) 1. \( x \lor y \rightarrow y \lor x \) if \( x \equiv y \)

(b) 13. \( x \lor x \rightarrow x \)

(c) 3. \( \text{false} \lor x \rightarrow x \).

Without showing the individual stages as we did with the \( \land \) operator symbol, the final expression produced after all the rules have applied is given below:

\[
\lor[\land[a, b], \\
\land[a, b, c, d], \\
\land[a, b, d], \\
\land[a, c], \\
\land[a, c, d]]
\]

and in figure 4.9.
4.2.2.4 Standardization of Sub-expression Length

This stage realizes the (lengthy) PDNF representation of an expression. At the end of the simplification discussed above, all the $\wedge$-expressions are not guaranteed to be minterms. Those $\wedge$-expressions which are not minterms must be transformed to minterms, and this is the unique characteristic of the PDNF. Up to now, the number of distinct variables in the expression is still unknown. The stage starts by obtaining the list of distinct variables in the expression (this list will later be used to influence the way in which the expression is manipulated). The implementation is via a merge algorithm which accepts two lists and delivers one with distinct and sorted elements. Again continuing with our example, the execution of the algorithm produces the list:

$$[a,b,c,d]$$

The list is used to compare the list of variables in each $\wedge$-expression. Where sub-expressions do not contain the same variables as found in the surrounding expression, they are
expanded to form minterms. These manipulations are facilitated by the following set of rules:

\[
\begin{align*}
7'. \ x & \rightarrow x \land \text{true} \\
4'. \ \text{true} & \rightarrow x \lor \neg x \\
9. \ x \land (y \lor z) & \rightarrow (x \land y) \lor (x \land z) \\
5. \ x \land y & \rightarrow y \land x \quad \text{if } x \leq y \\
22. \ x \land \neg y & \rightarrow \neg y \land x \quad \text{if } x \leq y \\
1. \ x \lor y & \rightarrow y \lor x \quad \text{if } x \leq y
\end{align*}
\]

The rules labeled: 4' and 7', act in the opposite direction to that induced by the HOO. However, they will be used in their "proper" directions later.

The implementation of the derivation of the minterms uses a merge procedure. For any missing variable in an \( \land \)-expression, two minterms are produced. In general, if \( n \) variables are missing in a term, then it is replaced by \( 2^n \) minterms. This is indicative of how the length of a PDNF expression can grow. The expansion of the \( \land \)-expressions in figure 4.9, using (1), produces the PDNF given in figure 4.10, and the list form is:

\[
\land[\land[a,b,c,d],
\land[a,b,c,\neg[d]],
\land[a,b,\neg[c],d],
\land[a,b,\neg[c],\neg[d]],
\land[a,\neg[b],c,d],
\land[a,\neg[b],c,\neg[d]]
] \land [a\lor b, c, \neg[d]]
\]
Resulting from our normalization methodology, this expression is unique and represents the equivalence class of boolean expressions which are algebraically equivalent to our original expression:

\[ \left( \neg (a \land \neg b) \lor c \right) \land d \lor a \land (a \land (b \lor c)) \]

Therefore, if we are investigating the equivalence of two boolean expressions, we need only reduce each to this form. If the original expressions are equivalent then the resulting expressions are identical.

We will now explain in detail how the minterms are obtained from an \( \land \)-expression. In figure 4.9, let us consider the first \( \land \)-expression. The list of variables contained therein is: \([a,b]\). Therefore, the list of variable missing is: \([c,d]\). We now consider each variable in turn. For the variable \( c \), the merge algorithm will combine \([a,b]\) with \( \text{true} \) then expand \( \text{true} \) as \((c \lor \neg c)\). Which is equivalent to combining \([a,b]\) with \([c]\) and \([\neg c]\) separately producing the list of \( \land \)-expressions:

\[ (\land[a,b,c], \land[a,b,\neg c]) \]
The process is repeated with $d$ and each $\land$-expression produced previously from $c$. Considering $d$ and the first $\land$-expression, we have the minterms:

$$(\land[a,b,c,d], \land[a,b,c,\neg[d]])$$

and with the second, we obtain

$$(\land[a,b,\neg[c],d], \land[a,b,\neg[c],\neg[d]])$$

which all together gives an ordered set of the four minterms thus:

$$\{\land[a,b,c,d], \\
\land[a,b,c,\neg[d]], \\
\land[a,b,\neg[c],d], \\
\land[a,b,\neg[c],\neg[d]]\}$$

4.2.2.5 **Factorization of The PDNF**

Figure 4.10 has demonstrated how awkward the representation of the PDNF can be compared to the inherently simpler expression in figure 4.9, or the textual form of figure 4.2. This will become even worse as the number of the variables increases. Because of this, a more manageable form is sought: RDNF, which is obtained by factorizing the PDNF. It can be observed that, among the minterms some factors are repeated. Practically, in terms of space usage PDNF is very inefficient. A more efficient representation could be one in which every duplicated factors are represented by just one. Following this line of thought leads to the application of certain rewrite rules in the opposite directions. However, the applications are controlled by $\sigma$ extended by the
lexicographical order on the identifiers. For instance, consider the PDNF:

\[ V[∧[a,b],
    ∧[a,¬[b]]] \]

The expression has two (variable) identifiers \( a \) and \( b \). The factor \( a \) is common to the two minterms and, moreover, it is less than \( b \) lexicographically (i.e. \( aσ_{\text{lex}} b \)). Therefore, factoring it out will transform the PDNF to

\[ ∧[a,
    V[∧[b],
    ∧[¬[b]]] ] \]

The above expression, as we can notice, contains binary operators \( ∧ \) with only one argument each. We can simplify these further using the supplementary rule: \( ∧(x) \rightarrow x \), which then gives the expression below:

\[ ∧[
    a,
    V[b,
    ¬[b] ]
  ] \]

Notice that the factorization process pushes in the operator \( V \) and pulls out the operator \( ∧ \). If the PDNF and the factorized form are represented in their infixed forms and combined into a directed equation we have:

\[(a∧b)∨(a∧¬b) \rightarrow a∧(b∨¬b).\]
This corresponds to the distributive law (rewrite rule 9) but in the opposite direction and is therefore labeled as 9'.

An important constraint is that the factorization is driven by the lexicographical order of the identifiers. The factoring out of an identifier is performed on the common identifier which is lexicographically lower than all others. After each application, the remaining \( \land \)-expressions are checked. If they are all the same, the list is replaced by the singleton. And we are done. Otherwise, the factorization is recursed on the whole remaining \( \land \)-expressions. In some cases the variable identifier \( a \) may appear in two different forms, i.e. \( a \) and \( \neg a \). When this situation exists among the minterms/\( \land \)-expressions, the list is split into two - one consisting of those containing the identifier and the other, those containing its complement.

Assuming that we have a PDNF given by:

\[
\begin{align*}
&\lor[\land[a,b,c], \\
&\land[a,b,\neg[b]], \\
&\land[\neg[a],b,c], \\
&\land[\neg[a],\neg[b],c]
\end{align*}
\]

the list of minterms will split to give:

\[
\begin{align*}
&\lor[\land[a,b,c], \\
&\land[a,b,\neg[b]], \\
&[\land[\neg[a],b,c], \\
&\land[\neg[a],\neg[b],c]
\end{align*}
\]
The factorization is then performed in two parts, that is relating the $V$ to the first list and factor out the least identifier, and again, relating it to the second to factor out the corresponding complement. However, because of the split the $V$ is retained and new ones introduced in order to maintain a correct syntax. In addition to reducing the remaining list to the singleton when the same $\wedge$-expressions are obtained after a factorization process as discussed above, the remaining expressions for the two lists of minterms/$\wedge$-expressions are checked for equality. If the remaining expressions are the same, the process will return one of them as the result.
Illustrating this with the above example:

\[
V[\land[a, b, c], \\
\land[a, b, \neg[b]], \\
\land[\neg[a], b, c], \\
\land[\neg[a], \neg[b], c] \\
\].
\]

We have by factoring out \( a \) and \( \neg a \), the expression given below:

\[
V[\land[a, \\
V[\land[b, c], \\
\land[b, \neg[c]], \\
\land[\neg[a], \\
V[\land[b, c], \\
\land[b, \neg[c]] \\
\].
\]

We can see that \( V(\land(b, c), \land(b, \neg c)) \) is common to the \( \land \)-expressions. Then, by combining them the result of the factorization is:

\[
\land[V[ \\
\land[a, \\
\neg [a] \\
], \\
V[\land[b, c], \\
\land[b, \neg [c]] \\
}\]

The conjunction, $\vee(a, \neg a)$, in the above expression, is further transformed to the constant: $true$, given the expression:

$$\land[true, \\lor[\land[b, c], \land[b, \neg[c]]] \land[b, \neg[c]]]$$

Which, yet again, transforms to

$$\lor[\land[b, c], \land[b, \neg[c]]].$$

If we repeat the factorization process on the above expression we will obtain $b$ as the result, i.e.

$$\land[b, \lor[c, \neg[c]]] = \land[b, true]$$

$$= \land[true, b] \quad \text{(using } p\text{)}$$

$$= b.$$

In obtaining the above expression, the rewrite rules used are:

$$x \lor \neg x \rightarrow true$$

$$true \land x \rightarrow x$$

and

$$x \land y \rightarrow y \land x \quad \text{if } x \Rightarrow y$$

We can notice that the directions of the first two rules are (like the distributive law) in the opposite direction of their previous applications.
This then brings us to the set of rules that effect the factorization of the PDNF to RDNF as:

\[
\begin{align*}
9' &. \quad (x \land y) \lor (x \land z) \rightarrow x \land (y \lor z) \\
4 &. \quad x \lor \neg x \rightarrow \text{true} \\
7 &. \quad \text{true} \land x \rightarrow x \\
5 &. \quad x \land y \rightarrow y \land x \quad \text{if } xpy
\end{align*}
\]

The application of these rules to the PDNF given in figure 4.10, produces the RDNF we have below and in figure 4.11:

\[
\land[a, \\
\lor[b, \\
\land[\neg[b],c]].
\]

![Figure 4.11](image_url)

The infix representation of figure 4.11 is:

\[
a \land (b \lor (\neg b \land c))
\]

Notice the change in the operator ordering in figure 4.9, this is due to the re-orientation of the rewrite rules involved in the derivation.
4.3 Conclusion.
In the development of our TRS for the simplification of boolean expressions we considered a sufficiently large set of equations. This is because we required enough equations to enable us demonstrate our intention fully well. Although, our work is not primarily on the completion of a given set of equations, nevertheless, there are some problems involving an arbitrary set of equations. With respect to our type of TRS, some interrelated problems are:

1. is the defined standard form adequate,
2. does the failure of deriving a standard form depend on the set of equations,
3. is a given set of equations sufficiently complete to guarantee that the rewriting methodology can be fully applied?

These three problems revolve around one main point—sufficiency of a set of equations. Indeed they can be regarded as restatements of the same basic problem. Many schools of thought have studied the problem especially in the field of generating a complete set of equations (or rewrite rules) for equational theories (as briefly discussed in section 2.3.4). Research in this area is still continuing. Our contribution is in a different dimension. We are not proposing any new completion strategy. But in the chapter 6 and 7 we look on how to advise the user on the sufficiency of his set of equations.
In the chapter we traced a transformation of a simple example so as to illustrate how the segmented TRS works. The segmentation and an order in which the rules are applied at each stage gave the system's outstanding characteristic: determinism, which is lacking in the traditional TRSs. The flow chart in figure 4.12 refreshes our memory with the stages in the system. The segmentation also necessitated the application of some rules in different directions at different stages.

However, the implementation of our strategy is restricted to algebraic systems that have AC-operators. The AC property was essential to allow local restructuring at the level of each AC operator symbol.
Adjustment on the order of Operator Symbols

Restructuring at AC Operator levels

Simplification at AC Operator levels

Standardization of Sub-expressions to the same Length

Factorization of PDNF

Figure 4.12
5 A NEW STRATEGY FOR TERM-REWRITING SYSTEMS II: FORMAL PRESENTATION

We have just given a detailed description of our strategy applied to the familiar, and well understood, Boolean algebra. Of course little would be gained if this was the only algebraic structure to which our strategy could be applied. Fortunately this is not the case and we now give a formal description of how to deal with an arbitrary single sorted algebraic system. (In chapter 8 we shall consider extending the method, and its 'implementation', to multi-sorted algebras. In chapter 6 we give another example of a familiar, single sorted, algebraic structure. We purposely restrict the examples to structures which are well known and of great practical importance so that attention is concentrated on the TR methodology rather than distracted by details of the underlying algebra).

This chapter is in two main parts. Firstly, starting from suitable orderings on adequate set of algebraic equations, we give the formal description of the, layered, TRS. Secondly we itemize the necessary conditions required to ensure applicability of the method.

5.1 The Method

We limit our consideration to the manipulations of expressions which, implicitly, define all allowable manipulations of expressions based on a single data sort, say X. Hence an expression is well-formed if it is syntactically
well-formed, i.e. it can be parsed with the appropriate grammar. Additionally we should note that we are not primarily concerned with expressions whose evaluation fails (because of a division by zero or some such similar error) but merely preserving the essential properties of an evaluation; if evaluation of the initial expression succeeds then so should the evaluation of any derived expression.

We assume that each operation within the algebra is of the type \( x^n \rightarrow x \) where \( n \geq 0 \).

(Operators that take three or more operands will generally be written in prefix form so as to circumvent complications at the grammatical level of description.)

Grammatically, all symbols used in the expressions are called operators. Restricting these (slightly) to those symbols representing conventional computational operators, constants (operators with zero operands) and identifiers (which represent "unevaluated" functions of type \( \text{Memory} \rightarrow X \)) we obtain the set \( \Theta \) of symbols which are used in semantic trees, as in figure 5.1:

![Figure 5.1](image-url)
Essentially we have merely disregarded those symbols used to impart the tree structure on a linear, textual, expression.

On $\Theta$ we now define a total order relation $\sigma$. We shall use $\sigma$ as the basis of precedence relations within the expressions. Hence it would seem not sensible to allow any symbols to be $\sigma$-related to constants or to identifiers. However, in order to make the ordering total - a property which we shall require later - we include, within $\sigma$, a component $\sigma_{\text{lex}}$ which orders those 'operators' lexicographically. Consequently the Hasse diagram associated with $\sigma$ is always of the form depicted in figure 5.2.
Since the initial expression is well-formed we can always use its parse tree, generated from a necessarily unambiguous operator precedence grammar, G (as described in section 3.1 of chapter 3), to obtain a corresponding semantic tree.

The three common forms of non-trivial tree components which are found in the parse trees (using A, B, C, ...) and their corresponding semantic trees (TA, TB, TC, ...) are shown in figure 5.3.
From a given expression we therefore pass via its parse tree to a bracket-free semantic tree. We shall henceforth, without loss of generality, regard an expression synonymously with its semantic tree. The set of these trees is therefore, modulo the inclusion of superfluous brackets (etc.), isomorphic with $\mathcal{L}(G)$.

It may be recalled that all expressions under consideration are generated by a grammar $G$. From $\sigma$, a total ordering on $\Theta$ ($\leq_T$), we can now define $\rho$, a total ordering on $\mathcal{L}(G)$. We do this by reference to the set of semantic trees, each of which represents an equivalence class of bracketed (perhaps superfluously 'over' bracketed) expressions in $\mathcal{L}(G)$. 
Given two distinct trees, we $\rho$-order them, using $\sigma$ to compare nodes in a hierarchical fashion, starting at the root; the symbol at the root denoting the final operation in the evaluation of the corresponding expression.

For $t, s \in L(G)$ where $t \neq s$, $t = \Phi(t_1, \ldots, t_n)$ and $s = \Psi(s_1, \ldots, s_m)$, we define $\rho$ by saying $t \rho s$ if:

1. $\Phi \sigma \Psi$ and $m, n \geq 1$.
   
   or

2. $\Phi = \Psi$ and there is some $i$, $1 \leq i \leq m$ and $1 \leq j \leq n$ such that $t_j = s_j$ for $j < i$, $t_i \neq s_i$ and $t_i \rho s_i$.
   
   or

3. $n \geq 1$ and $m = 0$ (meaning that $s$ is either a constant or an identifier)
   
   or

4. $t$ and $s$ are both constants or identifiers and $t \leq_{\text{lex}} s$ (where $\leq_{\text{lex}}$ denotes the lexicographic order with respect to $\rho$)
   
   or

   $t$ is an identifier and $s$ is a constant.

If $t \rho s$, we say that $s$ is 'better' than $t$. Moreover, when $t$ and $s$ are algebraically equal and $t \rho s$ then we may regard $s$ as a (structural) improvement of $t$ since $s$ is in normal form (or nearer to it).

From a given grammar $G$ (and hence a set of wffs), and a given ordering $\sigma$ on $T$, we have defined the ordering $\rho$ on $L(G)$. All
the manipulations that are allowed in transforming an expression \((e \in L(G))\) are given as equations. These will usually be axioms but we also include other, algebraically superfluous, equations which provide direct information relating to all legal operator combinations that can occur in the semantic trees of well-formed expressions. Exactly which combinations are desirable or otherwise is determined by \(\sigma\).

An equation is of the form
\[
e_1 = e_2
\]
where \(e_1 \in L(G)\) and \(e_2 \in L(G)\). Thus, provided \(e_1\) and \(e_2\) are not structurally equivalent (by which we mean that they are not represented by the same semantic tree), we can direct the equation using \(\rho\) since either \(e_1 \rho e_2\) or \(e_2 \rho e_1\), but not both. We can therefore replace the equation by the rewrite rule:
\[
e_1 \rightarrow e_2 \quad \text{if } e_1 \rho e_2
\]
or
\[
e_2 \rightarrow e_1 \quad \text{if } e_2 \rho e_1.
\]

Armed with \(G\), (and \(\sigma\) and \(\rho\)) and a set of \(\rho\)-directed rules we can now begin to normalize an arbitrary expression in \(L(G)\).

5.1.1 **Operator Re-ordering**
The aim of this phase is to achieve trees (expressions) in which the operators occurring in all paths, from leaf to root, are consistent with \(\sigma\).

If there is an instance where this is not the case then it can be fixed locally by application of one of the rules.
Of course, there may be more than one such situation and rewriting one 'violation' may actually create others. Such 'knock on' effects can not be avoided but since $\sigma$ is an order relation they cannot repeat cyclically. Nevertheless we wish to make the processing deterministic and to do this we stipulate a tree traversal strategy by which we search for 'improperly' structured subtrees.

The strategy used is top-down left-to-right, (i.e. depth first) and we search for an instance as depicted if figure 5.4.

Here $*_0, \ldots, *_m$ represent arbitrary, not necessarily distinct, elements of $\Theta$; $1 \leq n \leq m$

\[
*_i \sigma *_0 \quad \text{for } 1 \leq i \leq n-1
\]

and

\[
*_0 \sigma *_n
\]

(usually $m \leq 2$ and hence $n = 1$ or $n = 2$ at most).
We now apply a rule, a unique rule which starts with the structure as in figure 5.4 and replace it with something 'better' - the sense of $\rho$.

This change in the semantic tree will not affect its value but will change its structure. It will not alter anything occurring above $*_0$ nor will it adjust the subtrees with $*_1...*_n$ as roots (except perhaps $*_n$ itself). However it may cause the operator $*_0$ in the tree to be moved and hence result in the next transformation to be performed may be at a higher level, etc ... In any case the whole tree has to be examined after the rewrite in order to determine whether further manipulation is necessary and, if so, which rule to apply to which subtree.

5.1.2 Layer Simplification

One outcome of the first rewrite phase is that all instances of similar operators on a given branch are adjacent. When an (binary) operator is both associative and commutative such multiple occurrences can be grouped together and the corresponding subtrees organized into a more compact form. (This could also be achieved to a limited extent if the operation was associative but not commutative, but we shall not develop that situation.)

Before considering the detail we note two other, unrelated but useful, simplifications that we can take advantage of; provided, of course, that the appropriate rules are available in the algebra.
The first of these concerns the occurrence of multiple instances of the same operands. Algebraically this may arise when 'multiples' occur in situations which necessitate the use of constants to achieve reasonably concise representation, e.g.

\[ a + a + a = la + la + la = (l+1+1)a = 3a \]

Here it is a moot point whether 3a is 3*a since a may not be of type integer!! We also have the classic problem associated with equationally specified algebraic systems with large set of constants, not only should they all be introduced individually and explicitly, but something needs to be said concerning their inter-relationship.

Rule: \( 1a \rightarrow a, na \oplus ma \rightarrow (n \oplus m)a \).

We side-step this problem by assuming (implicitly) that the integers (or at least the natural numbers) exist together with their arithmetic.

Of course, with other operations, such as and (\( \wedge \)) and or (\( \vee \)) in the last chapter, idempotence effectively deletes multiple instances of like operands and this avoids the need to count in all but a trivial way.

Rule: \( x \oplus x \rightarrow x \).
The second situation which might occur relates to the existence of 'over-riding constants'. Suppose we have an arbitrary AC operation \( \odot \); with signature, \( X^2 \rightarrow X \) and, moreover, there is a constant \( C \in X \) where \( C \odot x = C \) for all \( x \in X \). The existence of such constant obviously allows for the possibility of simplifying an entire \( a_1 \odot a_2 \odot \ldots \odot a_{n-1} \odot a_n \) expression to \( C \) if one of the \( a_i \)'s is equal to \( C \). This (real) reduction is then caused by the rule \( C \odot x \rightarrow C \). Following on from this we may even seek out ways in which \( C \) might arise, i.e. are there rules of the form

\[ \ldots \odot \ldots \rightarrow C \]

Despite several obvious common instances, operator-related constants of this kind are rare. Their generation from 'arbitrary' values is similarly unusual but when they do arise the following pattern is often present:

\[ x \odot \%x \rightarrow C \]

where \( \% \) is monadic.

So, if \( C \) exists and rules of this form also exist we require that our AC shuffling of operands results in the correct positioning of \( x \) and \( \%x \) (whatever \( x \) might be, it is not necessarily a ground symbol) so that the rules can be directly applied and the simplification achieved.

We can use \( p \) to achieve this but notice that this is a rule derivation driven by the existence of a constant such as \( C \). At the \( \odot \) level we first require that \( \% \odot \odot \) so that simple terms for \( \odot \) can include trees where root is \( \% \).
Then we need a rule

\[ x \otimes \%y \rightarrow \%y \otimes x \quad \text{if } xpy \]

and this must over-ride

\[ x \otimes z \rightarrow z \otimes x \quad \text{if } xpz \]

when \( z = \%y \)

5.1.3 Sub-expressions Standardization

We now expand the lowest AC operator level so as to ensure that all terms (at this level) include a reference to all identifiers used anywhere in the overall expression. The manipulations used here are similar to, but subtly different from, those used in the previous phase.

To aid the discussion we use \( \Theta \) to represent the principal operator in question. We wish to extend the operand list of an \( n \)-place \( \Theta \) term to include an extra operand which is currently absent. And to do this we need an identity element for \( \Theta \), \( I_\Theta \), so that

\[ x \otimes I_\Theta = x \quad \text{for all } x \in X \]

and some monadic function \( \% \) so that, for arbitrary \( y \in X \),

\[ y \otimes \%y = I_\Theta, \]

where \( \otimes \) is a binary operator and, necessarily, \( \Theta \circ \Theta \).

We orientate these equations to give rules

\[ x \rightarrow x \otimes I_\Theta \]

and

\[ I_\Theta \rightarrow y \otimes \%y. \]

With reference to lists of identifiers occurring in (or absent from) the term we can then, by use of appropriate
commutativity and HOO rules achieve the following progression:

\[
(t_1 \otimes \ldots \otimes t_{n-1} \otimes t_{n+1} \otimes \ldots \otimes t_m)
\]

\[
\to (t_1 \otimes \ldots \otimes t_{n-1} \otimes t_{n+1} \otimes \ldots \otimes t_m) \otimes I_e
\]

\[
\to (t_1 \otimes \ldots \otimes t_{n-1} \otimes t_{n+1} \otimes \ldots \otimes t_m) \otimes (t_n \otimes \kappa t_n)
\]

\[
\to ((t_1 \otimes \ldots \otimes t_{n-1} \otimes t_{n+1} \otimes \ldots \otimes t_m) \otimes t_n)
\]

\[
\otimes ((t_1 \otimes \ldots \otimes t_{n-1} \otimes t_{n+1} \otimes \ldots \otimes t_m) \otimes \kappa t_n)
\]

\[
\to (t_1 \otimes \ldots \otimes t_{n-1} \otimes t_n \otimes t_{n+1} \otimes \ldots \otimes t_m)
\]

\[
\otimes (t_1 \otimes \ldots \otimes t_{n-1} \otimes \kappa t_n \otimes t_{n+1} \otimes \ldots \otimes t_m).
\]

Apart from the rules involving \(I_e\) we need

\[
x \otimes y \to y \otimes x \quad \text{when } xpy
\]

\[
x \otimes \kappa y \to \kappa y \otimes x \quad \text{when } xpy \text{ and }
\]

\[
x \otimes (y \otimes z) \to \ldots .
\]

In practice the transformation is achieved by conditional application of the rules, \(x \to x \otimes I_e\) and \(I_e \to y \otimes \kappa y\) to introduce the missing identifier. This is followed by rule to re-order the operators \(\otimes\) and \(\otimes\), e.g. the application of the distributive rule, \(x \otimes (y \otimes z) \to (x \otimes y) \otimes (x \otimes z)\). And finally, we re-arrange the operands of the AC operators so they are ordered with respect to \(p\).

We have now reached our intermediate normal form. Its uniqueness from the original expression is guaranteed by the deterministic way in which the transformations have been applied and equal length of the sub-expressions at the least AC operator. It can also be argued that two expressions which are algebraically equal should reduce to the same intermediate form. If we have \(\alpha\) and \(\beta \in L(G)\) (where \(\alpha = \beta\)
algebraically but $\alpha \neq \beta$ structurally) then since $\rho$ is a total order over $L(G)$, then either $\alpha \rho \beta$ or $\beta \rho \alpha$, but not both. Consequently one can be reduced to the other and they cannot both be in (intermediate) normal form.

The form now achieved is adequate for all proper mathematical purposes, however it will usually be needlessly cumbersome. For example in the Boolean algebra of chapter 4, a tautological expression which had 10 identifiers would have been expanded to include 1024 terms each with 10 factors. Our next, and final, transformation would reduce such an expression to 'true'.

5.1.4 Factorization
From the intermediate form we may seek to factorize an expression in many ways. To ensure well-definition of our simplification we shall again use our basic orderings $\sigma$ and, by reference, $\rho$.

The factorization process is defined recursively with reference to (i) pairs of AC operators that occur at adjacent levels within the normal form and (ii) the length of the operand list associated with a given operator. Again we shall use the notation $\otimes$ and $\oslash$ for arbitrary, but related, operators within the algebra under consideration.
If a particular expression only has a single level of operator(s), i.e.
\[ \otimes(\alpha_1, \ldots, \alpha_n) \]
then no further simplification takes place except when \( n = 1 \) in which case the non-monadic operator \( \otimes \) is deleted, i.e.
\[ \otimes(\alpha) \rightarrow \alpha. \]

In the general situation we have more AC operators in the hierarchical description of the expression and we can work through the hierarchy starting at the bottom, i.e. with the (AC) operator \( \otimes \) which occurs first in the \( \sigma \) ordering of those operators in the expression. If \( \otimes \) is the next AC operator such that \( \otimes \sigma \otimes \) then we have:
\[
((\ldots \otimes \ldots) \otimes (\ldots \otimes \ldots)) \ldots (\ldots \otimes \ldots). 
\]
Or, in the prefix form:
\[
\otimes(\otimes(\ldots), \otimes(\ldots), \otimes(\ldots), \ldots, \otimes(\ldots)). 
\]

Again, if the \( \otimes \) operand list is of length one, we can discard the operator, so,
\[
\otimes(\otimes(\ldots)) \rightarrow (\ldots) 
\]

In the more general case we have a larger list of \( \otimes \) operands each of which is a \( \otimes \) operator, and the operand list of those operators are all equal in length. Moreover, the operands are identifiers or monadic operators applied to identifiers and they are ordered in accordance with \( \rho \).
Suppose, for ease of description, that the first operand is based on the identifier $a$ and the feasible monadic operators are $\mathcal{N}_1, \ldots, \mathcal{N}_p$. Thus, we may have:

$$\theta(\theta(a, \ldots), \ldots),$$
$$\theta(\mathcal{N}_1(a), \ldots),$$
$$\theta(\mathcal{N}_2(a), \ldots),$$
$$\theta(\mathcal{N}_p(a), \ldots).$$

Firstly, we bring out the leading common factors.

$$\theta(\theta(a, (\theta(\ldots), \ldots), \ldots), \ldots),$$
$$\theta(\mathcal{N}_1(a), (\theta(\ldots), \ldots), \ldots),$$
$$\theta(\mathcal{N}_2(a), (\theta(\ldots), \ldots), \ldots),$$
$$\theta(\mathcal{N}_p(a), (\theta(\ldots), \ldots), \ldots),$$

i.e.

$$\theta(\theta(a, Wa),$$
$$\theta(\mathcal{N}_1(a), W_1a),$$
$$\theta(\mathcal{N}_2(a), W_2a),$$
$$\theta(\mathcal{N}_p(a), W_pa)).$$
If $W_a = W_1 a = \ldots = W_p a$, then we can re-group as

$$\Theta(a, \mathbb{N}_1(a), \mathbb{N}_2(a), \ldots, \mathbb{N}_p(a)) \Theta W_a.$$ 

Here the $\Theta$ expression might simplify but in any case $W_a$ (and, in general, $W_i a$) will contain factors comprising $\Theta$ and an operand list of length one less than before. Hence we proceed by factorizing these shorter lists using the same technique.

The rules needed to achieve the factorization comprise the following:

$$\Theta(\Theta(\alpha, \beta), \Theta(\alpha, \delta)) \rightarrow \Theta(\alpha, \Theta(\beta, \delta))$$

and

$$\Theta(a, \mathbb{N}_1(a), \mathbb{N}_2(a), \ldots, \mathbb{N}_p(a)) \rightarrow I \Theta \quad \text{(if available)}$$

The first rule is necessary in order, that the factorization process can commence. The second and the third rules may follow respectively depending on their availability as mentioned earlier. Also needed, are the rules of the form $\Theta(\alpha) \rightarrow \alpha$ which simplify the expressions involving the binary operator $\Theta$ when left with a single operand. Notice that such rules are not derived from the original (algebraic) equations since the need for them is a direct consequence of using variable arity prefix notation for the AC operators. It is therefore merely a technical device to overcome a problem introduced by this notation.
5.2 **Formal Requirements**

We now consider the requirements for an algebra to be suitable for processing by our TRS. These divide into two groups: those relating to the underlying operations and those concerning equations.

5.2.1 **Operators**

Given a set on which certain operators are defined it may well be the case that one operator can be expressed in terms of others. It might also be the case that, for convenience or for other technical reasons (see chapter 6), we wish to introduce a new operator. In this situation the new operator has to be defined in terms of those already in the system. [Of course, these definitions are given as equations. Hence we cannot fully separate the investigation of operators from the equations or rules. However, we do need to investigate how the underlying order, $\sigma$, works before extending it to $\rho$ which is then used to direct the equations.]

These two scenarios give rise to the distinct possibilities that some expression could be 'normalized' in two distinct (but algebraically equivalent) structures, each of which is hierarchically consistent with $\sigma$ but which use different operators.
Essentially we have two questions that must be asked of each operator present in the $\sigma$-ordering that we use to define the normal form.

(i) Does the removal of the operator mean that some expressions cannot be normalized?

(ii) Does retaining the operator cause ambiguity in normalization?

Question (i) can be addressed by the techniques described in chapter 6 and 7. Gaps in the set of expressions that can be processed may not cause any great loss if the missing expressions do not arise naturally in the context of a given application. Plugging the gap by the (re-) introduction of the operator might, of course, lead to the situation prompted by the second question.

Question (ii) requires that we consider ambiguity and within the realms of context-free languages, such matters are generally undecidable. It may even be, as hinted above, that a certain operator is needed so as to allow normalization of some expressions and yet this same operator leads to ambiguity in the normalization of others.

We take a pragmatic, constructive, approach to the overall problem. Start with a single operator and include others only when it can be shown (again using the methods of later chapters) that the inclusion extends the set of expressions that can be processed. This is not fool-proof but, given a small target set of operators, the tools of chapter 7 make it
feasible to adopt this strategy with all operator sequences from that set. Although this cannot guarantee the absence of ambiguity it does at least mean that it is not introduced without any accompanying benefit.

5.2.2 Rewrite Rules
The major concern is that rules derived from the given equations should be adequate to support our term-rewriting methodology. The requirements for support of each stage of the method are discussed in the subsections that follow.

However, we must also look at the problem of ambiguity/contradiction within the rewrite rules. Such characteristics may not give rise to failure since the associated set of rewrite sequences may satisfy the Church-Rosser property (see chapters 4 and 9 for details) or just involve commutative reductions. Nevertheless in these cases no harm is done by forcing the reductions to be deterministic.

We consider two specific types of situation. Firstly, suppose we have two equations that involve the same expressions, e.g.

\[ A = B \quad \text{and} \quad A = C \]

where \( A, B \) and \( C \) are all syntactically different. Now \( \rho \) is a total order on such expressions and so suppose \( A \rho B \rho C \). We can then replace the two equations by the two rules:

\[ A \rightarrow C \]

and

\[ B \rightarrow C. \]
This merely short-circuits the reductions \( A \rightarrow B \) and \( B \rightarrow C \) but in so doing removes the non-determinism caused by having rules with identical lhs's.

The second case has already been alluded to in chapter 4 and involves rules with overlapping lhs's. O'Donnell [O'D 77], stipulates that such rules be prohibited since they may give rise to ambiguous results; but they need not. Guided by the 'equal lhs's' situation as discussed above and at the same time (i) attempting to remove an ambiguity at the cost of excluding some valid computations while (ii) trying to detect when such a potential ambiguity might be inherent to the original set of equations. In doing this we also expose another essential rule ordering maxim which is associated not with lhs equality or overlapping but generalization. We deal with this first.

Suppose that \( A_x \) represents an expression in which \( x \) is an arbitrary sub-expression of type \( X \); then any special expression can be substituted for \( x \) and rules with the resulting expression as its lhs must be applied, where applicable, before any (necessarily unique) rule with \( A_x \) as its lhs. For example, in the Boolean case studied in the previous chapter; the rules

\[
\text{true} \land x \rightarrow x
\]

must be attempted before

\[
x \land y \rightarrow y \land x
\]

when \( xpy \) since \( \text{true} \in \mathbb{B} \).
Now for the general 'overlapping' case. Using a variation on the above notation to include two sub-expressions suppose we have an expression $xA_y$ with two related rules:

$$xA \to B$$

and

$$A_y \to C.$$ 

Obviously a clash occurs when we are presented with $\alpha A\beta$. The non-determinism can be resolved as follows: Since $\rho$ totally orders expressions, let $B \rho C$ (say) and use the rules:

$$\alpha A\beta \to C \quad \text{(first)}$$
$$\alpha A \to B$$
$$A\beta \to C \quad \text{(remaining two in either order).}$$

Additionally, without loss but with the possibility of detecting an error, we can include

$$B \to C.$$
If B and C are equivalent this reduction is harmless; in other situations it yields a partial decision procedure for ambiguity. In such situation it can identify situations in which one constant transforms to another to another !!!. Resolving the example of chapter 4 compels us to replace the rules:

\[ \text{if } (x, l) \rightarrow 1 \]

and

\[ \text{if } (0, y) \rightarrow 0 \]

by

\[ \text{if } (0, l) \rightarrow 0 \]
\[ \text{if } (x, l) \rightarrow 1 \]
\[ \text{if } (0, y) \rightarrow 0 \]

and

\[ 1 \rightarrow 0. \]

We can now turn our attention to the examination of the kinds of rules that are necessary for our multi-stage normalization to succeed

5.2.2.1 **Operator Re-ordering**

This is the most fundamental step in our methodology and one which imposes the greatest demand on the underlying set of equations.

Given an ordering, \( \sigma \), on the set \( \Theta \), we need to be able to improve (and hopefully make acceptable) any hierarchical combination of operators from \( \Theta \) found in an arbitrary expression. Explicitly if we have a tree segment as in figure 5.5, where \( (\beta, \alpha) \Phi \sigma \), then we need to change it somehow.
At the very least we need a rule whose lhs, in the prefix notation, is of the form:

\[ \alpha(..., \beta x, ...). \]

Ideally the rhs of the rule will not include a similar conflict with \( \sigma \) but it is sufficient if the structure of the associated tree is 'improved' with respect to \( \sigma \) and that such improvements are well-ordered.

For a given \( \sigma \), and hence a target normal form, and the set of expressions we wish to process, gaps in the set of equations/rules can be determined as detailed in subsequent chapters.

5.2.2.2 *Layer Simplification*

In the most amenable situations the key operators within the hierarchy are AC and have 'over-riding constants'. These operators are those in terms of which other subsidiary operations may be expressed. (The selection of suitable operators and the replacement of certain operators by combinations that include those with AC properties is discussed in chapter 6.)
None of the requirements is absolutely necessary; we can cope with non-associative and non-commutative operations. The presence of AC operators simply reduces the number of normal forms and eases manipulations. The kind of equations that are described are of the form:

\[ \alpha \Theta (\beta \Theta \gamma) = (\alpha \Theta \beta) \Theta \gamma \]
\[ \alpha \Theta \beta = \beta \Theta \alpha \]
\[ \zeta \Theta \alpha = \zeta \]

where \( \alpha, \beta \) and \( \gamma \) are arbitrary sub-expressions and \( c \) is a constant. Equations that cope with multiple instances of like operands might also be present. E.g.,

\[ \alpha \Theta \alpha = \alpha \]

or

\[ \alpha \Theta \alpha = 2\alpha \]

Again, these may be convenient but are not necessary. Indeed if integers are introduced as above then we need to extend the formal system to deal with these also. (In practice we seldom do this and such manipulations will not be discussed here.)

5.2.2.3 Sub-expressions Standardization

The expansion of sub-expressions to include references to all appropriate identifiers depends on the existence of identity elements and their formulation based on an arbitrary identifier (whose value might be restricted so as to allow the desired combination).
Explicitly, at the $\oplus$ layer, we need 

$$\alpha \oplus I_\phi = \alpha$$

and, if available,

$$\beta \oplus K \beta = I_\phi$$

for a suitable $\beta$ and $K$ and where $\oplus \phi \phi$. Additionally, to facilitate re-organization to achieve the explicit sub-expressions required we need rules:

$$\alpha \oplus \beta \rightarrow \beta \oplus \alpha \quad \text{when } \alpha \rho \beta$$

$$\alpha \oplus K \beta \rightarrow K \beta \oplus \alpha \quad \text{when } \alpha \rho \beta \text{ and }$$

$$\alpha \oplus (\beta \oplus \gamma) \rightarrow \ldots \ldots$$

\[5.2.2.4 \text{ Factorization}\]

In essence all that is required here is (left) distributivity, driven backwards.

$$(\alpha \oplus \beta) \oplus (\alpha \oplus \gamma) \rightarrow \alpha \oplus (\beta \oplus \gamma).$$

Additionally, in order to cope with the new combinations that can arise, any derived identity elements need to be accommodated and, if possible removed. Of course the expression cannot be reduced to the empty string.

The kinds of rules acceptable here are:

$$(\alpha \oplus \beta) \oplus (\alpha \oplus \gamma) \rightarrow \alpha \oplus (\beta \oplus \gamma) \quad \text{(left distributivity)}$$

$$(\alpha \oplus \gamma) \oplus (\beta \oplus \gamma) \rightarrow (\alpha \oplus \beta) \oplus \gamma \quad \text{(right distributivity)}$$

$$\alpha \oplus K \alpha \rightarrow I_\phi$$

$$\alpha \oplus \beta \rightarrow \beta \oplus \alpha \quad \text{when } \alpha \rho \beta$$

and

$$\alpha \oplus I_\phi \rightarrow \alpha$$

$$I_\phi \oplus \alpha \rightarrow \alpha$$
A typical 'ideal' reduction would then proceed thus: we first left factor the leading terms to get
\[(\alpha \oplus (\ldots) \oplus \ldots \oplus (\beta \alpha \oplus (\ldots)\)]

If this factorization yields identical residues then we can use right distributivity so that

\[(\alpha \oplus \gamma) \oplus (\beta \alpha \oplus \gamma)\]

becomes

\[(\alpha \oplus \beta \alpha) \oplus \gamma\]

which might then simplify further to

\[I \oplus \gamma\]

and hence \(\gamma\). The same reduction is then applied to each of the, strictly smaller, sub-expressions that result.

5.3 Conclusion

In this chapter we have given a formal description of our proposed term-rewriting strategy. In particular, it generalizes that given in chapter 4 and is expressed in general algebraic terms. We have explicitly identified the requirements necessary for our methodology to proceed; these are followed up in the next two chapters. There has also been brief discussion of other properties that the algebra might possess and which could conveniently be used to simplify the representations. These, however, are not strictly necessary.
6. ON THE INADEQUACY OF EQUATIONS

Algebraic axioms are equations expressing the fundamental identities of a particular algebraic system. When using these as a term-rewriting system, they are directed to form rewrite rules to avoid the inherent non-termination property of the equations due to their symmetric nature. As already mentioned, we base the orientation of the equations on the operator ordering, defined in a certain normal form. In the following two sections, we discuss how these equations may be inadequate with respect to a sought-after operator ordering and its associated normal form. We shall also consider how to cope with non-AC systems and the chapter will include some examples illustrating our discussion.

6.1 On The Operator Ordering of The Target Form

Equations are required to be directed with respect to the operator ordering in the target form and for a set of operators a number of orderings is possible. Among these, some are definitely more favourable than the others. By this we mean, that their derivation may prove more fruitful than the others. On the other hand, since we cannot change how the underlying algebra works, some ordering may just be impossible to derive. In particular, orienting equations by reference to the top operator symbols, might not give useful rules if the internal structures of the sides of the equations are incompatible with \( \sigma \). Also, an attempt to obtain a normal form, may sometimes lead to an infinite
computation - non-termination. We suggest how this might be handled.

6.1.1 Operator Ordering Inadequacy

The directing of an equation by $p$ (the HOO), as shown in chapter 3, is achieved by considering the top operator symbols of the lhs and rhs or those of their subterms. Indeed we did not concern ourselves with the order of the operator symbols within the two sides. When this approach is sufficient it means that checking a set of equations (rules) and the subsequent manipulation is easy. However, this does not always suffice. For example, given an equation:

$$\mathcal{F}(\mathcal{H}(x), \mathcal{H}(y)) = \mathcal{H}(\mathcal{G}(x, y))$$

and an ordering on the operator symbols as:

$$\mathcal{F} \circ \mathcal{H} \circ \mathcal{G}$$

the derived rule is:

$$\mathcal{F}(\mathcal{H}(x), \mathcal{H}(y)) \rightarrow \mathcal{H}(\mathcal{G}(x, y)).$$

Although the rule is obtained, notice that its application achieves nothing. Within the term $\mathcal{H}(\mathcal{G}(x, y))$, towards which the equation is directed, the order on the symbols $\mathcal{H}$ and $\mathcal{G}$ which is $\mathcal{G} \circ \mathcal{H}$, does not conform with the desired order $\mathcal{H} \circ \mathcal{G}$. In essence, the rule is useless and obviously indicates that the ordering cannot be achieved for the expressions having the same form as $\mathcal{H}(\mathcal{G}(x, y))$. 
Examples:-

1. Consider the Boolean algebra where the operator ordering is:
   \[ \land \sigma \rightarrow \lor \sigma \lor \]
   This ordering is impossible to obtain for every boolean expression. In particular, consider the DeMorgan's law:
   \[ \neg(x \lor y) = \neg x \land \neg y. \]
   Neither of the sides favours the ordering, although the equation can be oriented to the left since \( \land \sigma \rightarrow \). The lhs and rhs produce the orderings \( \lor \sigma \rightarrow \) and \( \neg \sigma \land \) respectively and neither conforms with the ordering required. The inability to use the oriented equation successfully indicates that any expression (or part of it) which is of the form of any the sides of the equation cannot be transformed in the desired way.

2. In the same way, it can be shown that the ordering, \( \lor \sigma \rightarrow \sigma \land \), is inadequate by considering the other DeMorgan's law:
   \[ \neg(x \land y) = \neg x \lor \neg y. \]

3. Also the ordering, \( \sigma \lor \sigma \land \sigma \rightarrow \), is unacceptable since, again, it does not favour the two DeMorgan's laws.

6.1.2 Operator Set Inadequacy
A non-achievable operator ordering may sometimes be attributed to the operator set. If there is a reasonable number of operator symbols, we will expect more
interrelations among them and more likelihood of admitting several alternatives.

Consider a normal form of an algebraic system in which not all the operator set is involved. This means that among the rules there are some that eliminate those operators not mentioned in the normal form. There may be some expressions of the algebra which can never transform because of the constraint on the operator symbols in the normal form.

To transform these expressions (i.e. obtain a sufficient set of equations), we must include the operators into the normal form. It is only then that the rules required for the introduction of the operators can be sought.

6.1.3 Inadequacy of The Target Form

An attempt to obtain a required form may lead to infinite computation - and this violates the non-termination property of the underlying TRS. Here, non-termination is due to the fact that every expression produced is not in the required form, and the process goes on indefinitely trying to obtain an equivalent expression in the desired form. In particular, when there is a related sequence of the form:

\[ t_1 \rho t_2 \rho \ldots \rho t_i \rho \ldots \rho t_j \rho \ldots \]

and \( t_j \rho t_i \), for some \( i \) and \( j \) and \( j > i \). This means that the sequence from \( i \) to \( j \) repeats. Such a 'loop' is the likely cause of non-termination. However, a sequence of relations need not loop to be non-terminating. A more general property
than looping which detects the non-termination of some relations is 'self-embedding'.

**Definition** A term $s$ is *homeomorphically embedded* in a term $t$ if $s$ may be obtained from $t$ by deletion of some symbols.

**Definition** A sequence of relations:

$$t_1 \rho t_2 \rho \ldots \rho t_i \rho \ldots$$

is *self-embedding* if $t_i$ is homeomorphically embedded in $t_j$, where $j > i$ for some $i$ and $j$. Dershowitz [Der 82] claims that non-termination allows self-embedding.

To illustrating self-embeddingness, let us consider expressing the expression: $(x + y)^{-1}$ as sums ($+$) of products ($*$) of variables or inverse ($^{-1}$) of variables. Obviously, the operator ordering,

$^{-1} \sigma \ast \sigma +$, is associated with the grammar having productions:

- $E \rightarrow E+T|T$
- $T \rightarrow T*P|P$
- $P \rightarrow Q^{-1}|Q$
- $Q \rightarrow (E)|x|y$.

A possible contender is a rewrite rule which expresses the binomial expansion formula which is given by:

$$(x + y)^{-1} \rightarrow \binom{-1}{r} x^{-1} \ast y^r + (x + y)^{-1-r}$$

where $r$ is a positive integer and $C$ denotes the combination process. However, this produces an indefinite representation of the expression, $(x + y)^{-1}$. The indefinite representation is due to embedment of $(x + y)^{-1}$ in the rhs of the rewrite rule.
If \( r = 1 \), we have the rule as:

\[
(x + y)^{-1} \rightarrow (-1) \ x^{-1} \ast y + (x + y)^{-2}
\]

where \(-1C1\) is equal to \(-1\), the coefficient of \(x^{-1}\).

Alternatively, this can be represented as

\[
(x + y)^{-1} \rightarrow (-1) \ x^{-1} \ast y + [(x + y)^{-1} \ast (x + y)^{-1}],
\]

which makes the embedding of \((x + y)^{-1}\) in the rhs of the above rule obvious. Actually, the embeddingness shown in this example is a repetition of the lhs in the rhs of the rule. And as long as this is true the process will never terminate.

One important observation is that, a definition and introduction of a new operator will help to resolve the situation above. If we recall, in the example given above, it can be noticed that allowing quotient by introducing the operator symbol, '/', and including the obvious equation from the definition:

\[
x^1 \triangleq \frac{1}{x} \quad \text{if } x \neq 0
\]

we are sure of producing a definite term. Introducing new operators can be very helpful especially when they allow greater use of associated AC operators, but do not materially affect the mathematics. This modification will give the class of expressions of the form:

\[
(x + y)^{-1}
\]

definite representations.
A new operator ordering may then be given as:
\[ \text{id} \sigma \subseteq \sigma \cdot \sigma \cdot \sigma + \sigma / . \]
where \( \cdot \) denotes exponentiation to a positive power. The associated grammar is then:

\[
\begin{align*}
E & \rightarrow F/F \\
F & \rightarrow F+T|F-T|T \\
T & \rightarrow T*P|P \\
P & \rightarrow Q**N|Q \\
Q & \rightarrow (E)|\sigma|\text{id}.
\end{align*}
\]

The ordering from the above grammar can normalize any expression in field, but the operator ordering, \( \cdot \sigma \cdot \sigma + \), can only normalize a strict subset. Furthermore, the operator ordering, \( \cdot \sigma + \sigma \cdot \), since \( + \) is not distributive over \( * \), allows the normalization of an even smaller set of expressions. For an illustration, the equation \( x^*(y+z) = (x*y)+(x*z) \) will orient to the left and the class of expressions of the form:

\[ x + (y * z) \]

where \( y \) or \( z \) is not a factor of \( x \), will never transform. This is not the case for the operator ordering \( \cdot \sigma \cdot \sigma + \). An important remark worth noting with respect to this example is that, axioms like distributivity, need to be taken into consideration when choosing an operator ordering for a normal form. More discussion on the set of expressions that can be normalized by a set of equations (rules) is given in chapter 7.
6.2 Insufficient Sets of Equations

An inadequacy of a set of equations occurs, basically, when the equation needed to transform an expression (or part of it) is not contained in the set.

In particular, consider the boolean algebra. It is true that every boolean expression can be expressed as \( \text{sums}(V) \) (or \( \text{products}(\Lambda) \)) of \( \text{products}(\Lambda) \) (or \( \text{sums}(V) \)) of constants or variables or complement(\( \neg \)) of variables. This gives the operator ordering:

\[ \neg \sigma \to \sigma \land \sigma \lor (\text{or } \neg \sigma \to \sigma \lor \sigma \land) \]

But if the equation that transforms the order on the pair of operators: \((\neg, \Lambda)\) is missing, obviously, some expressions cannot be transformed to conform to that ordering.

We now give some guide-lines which, if observed when drawing up set of equations, improve the possibility of sufficiency. It may be recalled that the target form defines an operator ordering, the intention being to orient the equations and use them in an attempt to achieve the appropriate operator ordering within all expressions: This suggests that there should be some property required of the equations, so as to make their usage worthwhile.

Obtaining a target form involves the re-ordering of the operators in the semantic tree. So, we first need to look at the necessary relationship between the rhs and lhs of an equation with respect to their top operators. An equation may be viewed as the definition relating the top operator symbol on the lhs to the top operator symbol on the rhs, and...
vice versa. If, for every pair of operator symbols in a set, there is an equation expressing this condition, the reordering of the operators in an object tree (which causes a change of the top operators of the subtrees) is assured. But this condition is excessively restrictive.

An alternative way to reason about this is by considering the orders on pairs of operators. In an object tree different orders are possible, i.e. by considering an operator and its parent or its siblings. Those orders not in agreement with those of the target tree, need to be transformed. Now to the equations. It is expected that among the equations there should be at least one side which represents/matches the unacceptable order in the object tree. Where all the unacceptable orders are not represented, this signals an insufficiency of the equations for the transformation of the object tree.

For instance, consider a set of operators: \{F, G, H, C\}, where F and G are binary operators, H a unary operator and C a constant. Assuming that in the target tree the order on the operators is:

```
    F
   /|
  G / |
 /  H
 /   C
```

(Implying the order the symbols F, G, H and C is required in the target tree.)
Again, assuming that a part of an object tree looks like figure 6.1, the parts marked (1) and (2), indicate unacceptable orders on the operators.

Figure 6.1

To put these right, we require equations with their lhs's corresponding to these orders and their rhs's giving the acceptable orders. However, the set of the unacceptable orders of an algebraic system, can be easily obtained from the set of the operators. Assuming that the operator set given above is for an algebraic system and also, the above ordering is defined by a standard form, we can then form the order table as in figure 3.2.
Notice that the marked orders in the above tree are included in the figure and are shown as unacceptable orders. If there is no equation for every unacceptable order, then the set of equations is insufficient to drive the transformation of every expression of the algebra.

6.3 **Non-AC Operations**

When planning out the stages for a system, it is essential to examine the rewrite rules and identify the embedded properties of the operator symbols. This information may even suggest what can or cannot be done with the rewriting system of the rules. For instance, if some operator symbols are both associative and commutative, a lot can be achieved (as shown in chapter 4). Very little can be achieved if
there is no operator symbol exhibiting either or both of these properties.

In particular, if an arbitrary operator symbol, $\ast$, is neither associative nor commutative, then the expressions:

```
  *  *  *
 / \ / \ / \\
x   y z   x   y z
```

and

```
  *  *  *
 / \ / \ / \\
x   y x   y   x
```

cannot be transformed further, and we are done. Each expression is already in a non-transformable state and in particular each pair of the trees are not algebraically equal. In order to accommodate this situation (non-associative and/or non-commutative operators) we must allow larger numbers of normal forms. Explicitly, we may accept all the above trees as 'normal'.

In some special situations there could be a solution. For instance, consider the operator symbols $+$ (plus) and $-$ (minus). Plus is commutative and minus is not. Examine the expressions $(a - b)$ and $(b - a)$. If we have a standard form in which $a$ must be considered before $b$ (because of their lexicographic order) then the form of $(b - a)$ needs transforming. With respect to the minus nothing can be done. However, rewriting the expressions using the commutative plus might help achieve the desired order.
In particular, rewriting
\[ a - b \quad \text{as} \quad a + (-b) \]
and
\[ b - a \quad \text{as} \quad b + (-a). \]
Now we can use the commutative property of plus to rewrite
\[ b + (-a) \]
further to
\[ (-a) + b. \]
Representing these in tree forms we have:

```
  +     +
  / \   /  \
 a -  b a b
  |    |  |
  b   a
```

which are of the same standard form. A similar re-definition is possible to cope with the non-commutative division operation. Also, in the boolean algebra, the non-commutative difference operation, denoted by ' \', can be defined in terms of \( \wedge \) and \( \neg \), i.e.:

\[ A \setminus B \equiv A \wedge \neg B \]

Note that the reason why we could make these replacements is because of the relationship between the operators involved. We have seen the possibility of defining a non-commutative operation in terms of a commutative and a subsidiary monadic operations. However, this type of re-definition may not be possible for every non-commutative operation.
6.4 Conclusion

We have discussed the main causes of the inadequacy of a set of equations, i.e. how the desired orderings on the operator symbols in the target form and the set of available equations cause normalization to fail. We also indicated how introducing and including new operators may help resolve the situation. Failure might also necessitate re-formulation of normal form. When examining the set of equations it is essential to see that all the possible orders between the operator pairs are represented and can, if required, be changed appropriately.

The requirement that all expressions can, after a finite number of rewrites, be transformed into an acceptable 'normal' form is crucial to our term-rewriting strategy. The next chapter is devoted to isolating the situations when such rewrite sequences cannot be found within a given set of equations relative to a pre-defined normal form.
7 PRACTICAL CONSIDERATIONS

We discuss in this chapter how we can mechanically check if a set of equations is large enough to allow the manipulations we require, i.e. if every expression of the algebra can be transformed using the set. In any case the methodology produces the subset of expressions for which the equation set is sufficient. In section 7.1 we present how information about the algebra in question may be given. This is an algebraic specification consisting of sort, operations and equations (defining the identities or properties) of the algebra. In section 7.2 we show the generation of forms (combinations of the operators) from the specification. The sections 7.3 through 7.6 go on to show how the relation obtained from the equations in the specification is analysed. The relation and the forms are used to form a matrix and some calculations on this matrix provide a partial solution to the sufficiency of the TRS consisting of these equations. Failure signifies an "algebraic" gap in the set of equations, and the state of the matrix then supplies the user with important information in case he wants to "complete" the set of equations. Because the matrix is going to be large for some arbitrary system we have to be careful with the matrix calculations and the problem of using unreasonably large amounts of space.

7.1 Algebraic Specification

The notion of defining an algebra in terms of operations and equations, was picked up by Zilles (see [EhMa 85]) in order
to specify abstract data types like stacks, queues and strings by algebraic equations. Sorts, constants and operator symbols are syntactical notions which are constituent parts of signature. An algebra of a given signature has a base set for sort, constant element(s) and operation associated to each constant and operator symbol respectively. Operators and their operands can be combined to define terms which may be evaluated in each algebra of a given signature. By adding equations to a signature we obtain an equational algebraic specification. An equational algebraic specification represents (by definition) all algebras which satisfy those equations.

We will give a finite syntactical representation, which is a specification of an algebraic system. There can be several different sorts and the declaration of operator symbols must be given explicitly. Here we will give the specification of an algebraic system having only one sort.

In syntactical terms, the notion of "equation" is somewhat dangerous, because the terms are not at all equal syntactically (indeed that is the whole point of having such equations). For instance, consider associativity:

\[(x+y)+z = x+(y+z)\]

the lhs and rhs of the equation are not syntactically identical. But on the semantical level, if the variables \(x\), \(y\) and \(z\) are instantiated to terms of the algebra, say,

\[(b_1+b_2)+b_3 = b_1+(b_2+b_3)\]

the problem is contained.
We give a syntactical representation of an illustrative algebraic system (the specification of the algebra) as follows:

sort: \( S \)

ops: \( C \to S \)
\( \%= S \)
\( \mathcal{F}(S, S) \to S \)
\( \mathcal{G}(S, S) \to S \)

eqn: \( x, y, z \in S \)
\( \%= (x) = x \)
\( \mathcal{F}(x, \mathcal{G}(y, z)) = \mathcal{G}(\mathcal{F}(x, y), \mathcal{F}(x, z)) \)
\( \mathcal{G}(x, \mathcal{F}(y, z)) = \mathcal{F}(\mathcal{G}(x, y), \mathcal{G}(x, z)) \)
\( \%= (\mathcal{F}(x, y)) = \mathcal{G}(\%= x, \%= y) \)
\( \%= (\mathcal{G}(x, y)) = \mathcal{F}(\%= x, \%= y) \)

The set of terminals \( T \) associated to this algebra is obtained from the \( \text{ops} \) declaration of the specification. We only require the explicit inclusion of the special constants. For example, in the set of integers there are infinitely many constants but the only important ones are 0 and 1. The set \( T \) from the specification is:

\( T = (C, \%, \mathcal{G}, \mathcal{F}) \)

(Here \( C \) is a special constant.) \( T \) is always finite but the set of sentences \( L(G) \) will often be infinite. The arities of the operator symbols are important (as we will see later) and
these are also obtained from the ops declaration. The set of the symbols and their arities in the above specification is:
\{C(0), K(1), G(2), F(2)\}.

7.2 The Construction of Forms From Sets of Operators

The operators are required to have fixed arities as indicated in the specification. In particular, the constants are associated with the integer 0, the unary operators with 1, the binary operators with 2 and so on, the numbers corresponding to their arities. This is essential because the size of each form is governed by the arity of the top operator symbol. For example, a special constant standing alone represents a form and its arity is 0; if it is a top symbol it can not combine with another symbol. The algorithm 1 given below summarizes the procedure for constructing the forms.

Algorithm Outline 1:

If an operator symbol deliver a result of sort S
then

If its arity equal to n
then combine with n symbols delivering sort S.

Let us illustrate the construction of forms from the set:
\{C(0), K(1), G(2), F(2)\}

In the specification we have just one sort, so its test is irrelevant, but we must note the importance when there are several sorts. The generation of the forms involves considering the operator symbols in turn and combining with
considering the operator symbols in turn and combining with every symbol which can produce suitable data. The constant $C$ has a zero arity so, it is itself a form and can not combined with any other symbol. But in the case of $\&$ which has arity one, the set of forms in which $\&$ is the top operator symbol is given by:

\[
\{ \& (x) \mid x \in T \}. 
\]

Also, for the symbols, $\mathcal{G}$ and $\mathcal{F}$, their respective sets of forms are

\[
\{ \mathcal{G}(x, y) \mid x, y \in T \} \quad \text{and} \quad \{ \mathcal{F}(x, y) \mid x, y \in T \}
\]

respectively. The union of these sets and the singleton $\{C\}$ gives the set of forms generated from the symbols in $T$. Let the set be denoted by $\Sigma$, then,

\[
\Sigma = \{ C \} \cup \{ \& (x) \mid x \in T \} \cup \{ \mathcal{G}(x, y) \mid x, y \in T \} \\
\cup \{ \mathcal{F}(x, y) \mid x, y \in T \}
\]

If we substitute all the possible symbols for the variables $x$ and $y$, the set of forms will then be:

\[
\Sigma = \{ C, \& (C), \& (C), \& (\mathcal{G}), \& (\mathcal{F}), \mathcal{G}(C, C), \mathcal{G}(C, \mathcal{F}), \mathcal{G}(C, \&), \mathcal{G}(\mathcal{G}, C), \mathcal{G}(\mathcal{F}, C), \mathcal{G}(\mathcal{F}, \mathcal{G}), \mathcal{G}(\mathcal{F}, \mathcal{F}), \mathcal{F}(C, C), \mathcal{F}(C, \mathcal{F}), \mathcal{F}(C, \&), \mathcal{F}(\mathcal{G}, C), \mathcal{F}(\mathcal{G}, \mathcal{F}), \mathcal{F}(\mathcal{G}, \&), \mathcal{F}(\mathcal{F}, C), \mathcal{F}(\mathcal{F}, \mathcal{G}), \mathcal{F}(\mathcal{F}, \mathcal{F}), \mathcal{G}(\mathcal{F}, C), \mathcal{G}(\mathcal{F}, \mathcal{F}) \}
\]

We included $S$ in $\Sigma$ to stand for (other) arbitrary expressions such as identifiers of the sort $S$. 


7.3 Representation of Binary relations as Boolean Matrices

The ordering $\sigma$, defined on $T$ (obtained from an operator precedence grammar associated with $T$, as in chapter 3), is employed in the derivation of the HOO. $\text{HOO}$, as we ascertained (again, in chapter 3), is an order relation $\rho$ utilized in relating terms and hence, most importantly, in the orientation of equations. The oriented equations constitute the initial relation: The ordered pairs of forms from the oriented equations are obtained by removing the variables in the $\text{lhs}^3$ and $\text{rhs}^3$ of the equations. For instance, the oriented equation:

$$F(x,y, z) \rightarrow F(x,z), F(y,z))$$

generates the ordered pair of forms:

$$F(S) \rho F(F,F)$$

Here $S$ is either $\land$, $\lor$, $\implies$ or $\equiv$. Hence, from the above initial relation the following set of ordered pairs is obtain:

$$F(S) \rho F(F,F)$$

This is simply achieved by instantiating $S$ to each of the operator symbol.

Let us give an illustration with the directed equations from boolean algebra which are given as:

$$\land(x, \lor(y, z)) \rightarrow \lor(\land(x, y), \land(x, z))$$

$$\neg(\land(x, y)) \rightarrow \lor(\neg x, \neg y))$$
\[ \lnot (x) \rightarrow x \]

The sort boolean (bool) is denoted by \( \mathbb{B} \). It follows then that \( \mathbb{B} \) is any boolean expression or the top operator symbol of a boolean expression which can be anyone from the set:

\{false, true, \neg, \wedge, \vee\}.

Now, the initial relation obtained above directed equations is:

\[
\begin{align*}
\wedge(\mathbb{B}, \vee) &\rightarrow \vee(\wedge, \wedge) \\
\neg(\wedge) &\rightarrow \vee(\neg, \neg) \\
\neg(\neg) &\rightarrow \mathbb{B}
\end{align*}
\]

The \( \mathbb{B} \) in the first ordered pair can be either a boolean expression or the top operator. But the \( \mathbb{B} \) in the third ordered pair is restricted to only boolean expressions (and more, the expression must have been contained in the left term of the pair). Considering the set of operators, \{false, true, \neg, \wedge, \vee\}, the set of ordered pairs generated from the first ordered pair of the initial relation is:

\[
\{ \wedge(\text{false}, \vee) \rightarrow \vee(\wedge, \wedge) \\
\wedge(\text{true}, \vee) \rightarrow \vee(\wedge, \wedge) \\
\wedge(\neg, \vee) \rightarrow \vee(\wedge, \wedge) \\
\wedge(\wedge, \vee) \rightarrow \vee(\wedge, \wedge) \\
\wedge(\vee, \vee) \rightarrow \vee(\wedge, \wedge) \}
\]

(Note that instead of the first two ordered pairs in the set we can have the pairs \( \wedge(\text{false}, \vee) \rightarrow \text{false} \) and \( \wedge(\text{true}, \vee) \rightarrow \mathbb{B} \), to correspond to the rules used in chapter 4. But depending on the stage concerned we can still have the pairs as given in the above set.)
The second and the third ordered pairs in the initial relation stand as they are, i.e. no additional pairs are derived from them.

Continuing with the specification, if the ordering on the operator symbols is:

$t \circ \% \circ \top \circ \bot$

then, using this ordering we obtain the HOO which we employ to orient the equations in the specification and consequently from that the following relation is obtained:

$\top(\% \circ \top) \circ \top \circ \top$

$T(S, \bot) \circ \top(S, T, T)$

$T(S, S) \circ \top(T, S, T)$

$\top(T) \circ \top(\top, \top)$

$\top(\top) \circ T(\top, \top)$

We must point out here that although, the $S$ in the pair, $T(S, S) \circ \top(S, T, T)$, may represent either a boolean expression or the top operator, however the pair as it is, is misleading. The derivation of the pair goes far beyond what we have considered earlier, this is because $S$ is required to be duplicated in the trees which their roots are represented by the two $\top$s on the lhs of the pair. It is only then that it is valid to derive the pair, $T(S, S) \circ \top(S, T, T)$. It is due to factors (omissions) such as noted here that our sufficiency check is only a partial decision procedure.

By instantiating the $S$s in the relation we increase the number of the ordered pairs in the relation. These pairs
form the entries of a binary matrix. Note that the instantiation increased the entries in the matrix.

The matrix is indexed by the forms ordered in the same way, i.e. if a particular form indexes say column 4, the same should index row 4. An order is essential because without it a transitive closure calculation on the relation is meaningless.

One may wish to represent the relation by 1s and the non-related pairs by 0s. Given a relation, locating the entries of the matrix just involves searching through the rows and columns for the related forms. The intersections of these rows and columns comprise the entries. The relation $\rho$ is asymmetric i.e.

$$x \rho y \Rightarrow y \rho x$$

is not true. The asymmetric property of the relation is maintained by strict consideration from row to column.

7.3.1 Implementation

The size of the set of forms obtained from a relation in most cases are likely to be small compared to the size of $\Sigma$. For an $n \times n$ matrix (where $n$ is typically very large) a considerable large storage space is needed. Characteristically, only a small fraction of the space is used for the actual relation and the remaining large portion for the unrelated pairs. Since we are interested in performing some calculations on the relation alone this will amount to waste of storage. Hence, we thought of using a
sparse storage i.e. a sparse matrix [Knu 68] to hold the relation. Nevertheless, we stored all the indexes which helps immensely in the searching process.

We used a dynamic data structure and an illustration is given in figure 7.1.

![Figure 7.1]

The row/column numbers, apart from representing what they imply, each corresponds to a particular form. It is easier to use reference numbers for the forms rather than the actual forms. In the first node - the root, we used -1 for both the row and column number so as to distinguish it from other nodes. Similarly, the headers (indexes) contain -1 either in
the row number field (in the case of column headers) or in that of the column number (in the case of row headers). There is a cycle in the addressing between the headers, e.g.

![Diagram](https://via.placeholder.com/150)

The root holds the address of the first header, the first header holds the address of the second header, the second holds the address of the third and so on. The last header holds the address of the root which then completes a cycle. This form of addressing is applicable to both row and column headers. However, the cycle does not cause any problem because we can always use the -1s at the root as a delimiter. All the row headers (or column headers) have their initial addresses to the next columns (or rows) set to nil. Similarly, any of the node in the matrix i.e. an entry, may have both or one or none of its address fields set to nil depending on the circumstances. By setting these address fields to nil rather than to the addresses of their headers facilitates the insertion and deletion processes.

### 7.4 Computation of The Closure of The Matrix

This is the computation of the transitive closure of the relation represented in a binary matrix. The procedure for the computation is given in algorithm 2 below:
Algorithm Outline 2 (Warshall's algorithm [War 62]):

While the address to the next column header (i) not equal to -1 do

While the address to the next row (j) (from each column header) not nil do

If index: [j, i] of a node not equal (i.e. j ≠ i) then

Apply the logical or to the ith row and jth row, and replace jth row by the result.

Explanation of the application of the logical or contained in algorithm 2:

Searching through the ith and jth rows simultaneously, if there is no entry in the ith row then the jth row is unchanged. Otherwise, if the column number of the node in the ith row is less than that of the node in the jth row (or there is no entry in the jth row), then insert a node with the same column number as that in the ith row and the row number the same as j. If the column number of the node in the jth row is less or equal to that of the ith row do nothing. And continue the process.

On termination of algorithm the transitive closure matrix $M’$ is produced. The transitive and reflexive closure could be computed by using:

$$M’ = M’ + I$$

where $I$ is the identity matrix. However, in our case it is sufficient working with $M’$, and moreover the addition of the identity indicates cycles in rewrites (see section 7.4.1.1).
7.4.1 Interpretation of the Closure Matrix

The closure matrix (in non-trivial cases) contains far more entries than the original matrix. These entries are at certain areas of the matrix. These areas correspond to the portions of the columns indexed by the acceptable forms. A better picture is seen if, prior to the indexing of the matrix, the forms are arranged in such a way that the acceptable forms are all grouped together. If this is done, the scene will be such that a section of the matrix $M^+$ contains more entries than the rest.

Whichever is the case (— either arranging the forms in some order or not), the embedded interpretation of these entries is the identification of the forms (indexing the rows) that are transformable. Some gaps indicate the inability to relate some forms, and the expressions which contain these forms cannot be transformed.

The transposition of these entries infers that the subset of $L(G)$ consisting of the identified forms only are guaranteed to be normalized, with respect to the considered set of equations and normal form. In other words, the subset represents the language (algebra) for which the given set of equations (rewrite rules) is sufficient to serve as the TRS.

7.4.1.1 Relationship Between Closure Matrix and Rewriting

The closure matrix represents an algebraic system (having been constructed from a specification of an algebra). Now
consider an expression $E$ in the algebra. Certainly $E$ consists of some forms and let these be denoted by $F_1, \ldots, F_k$ and the associated 'normal' forms be given as $N = N_1, \ldots, N_m$. Obviously these forms, $F_1, \ldots, F_k$ and $N$, are contained in the matrix. $E$ may or may not rewrite to its normal form which is $N$.

The failure to transform $E$ means that there exists some form $F_i \in E$ such that

$$\rightarrow^+(F_i) \cap N = \emptyset.$$ 

This is identified in the matrix by either no entry on the row of the matrix indexed by $F_i$, or there is a sequence of entries but outside the range of $N$.

$E$ is successfully transformable if for every $F_i \in E$ one of the following holds.

(a) $F_i \in N$, in that case nothing needs to be done. In the matrix, $M^*$, there are no entries for these forms, but in the $M^*$ they can only relate to identical forms in which case the entries are within $N$.

(b) $\rightarrow^+(F_i) \cap N \neq \emptyset$. This is seen as a sequence of entries in the matrix with some of them within $N$.

(c) $(F_i \rightarrow^+ F_j) \land (\rightarrow^+(F_j) \cap N \neq \emptyset) \land \neg (F_j \rightarrow^+ F_i)$.

If there is just one sequence of entries in the matrix representing (c) then it can be said that (b) and (c) are the same. Nevertheless, there can be two sequences, in which case the one associated to $F_i$ terminates outside $N$, i.e. the last entry in the sequence identifies the form $F_j$ (indexing a column).
Then, there is another sequence of entries form $F_j$ (new row) that has some entries in $\mathbb{N}$. By the transitive property of $\rightarrow F_i$ is transformed to its normal form. It is essential that there should be no cycle within the second sequence and this is represented by $\rightarrow(F_j \rightarrow^+ F_j)$; this can be the case if $\rightarrow^+$ is not reflexive.

However, it is a contradiction if it can be obtained that $C_i \rightarrow^+ C_j$, where $C_i$ and $C_j$ are constants. More generally the system is contradictory if $C_i = C_j$ but $C_i \not= C_j$ (i.e. they are different constants but can be shown to be equivalent by application of the rewrite equations). The $\rho$-directed rewrites are not bound to show this.

### 7.5 Incomplete Algebraic Systems

In some cases it may be that the set represented by a given TRS is a proper subset of the algebra; meaning that there are some expressions of the algebra of which their transformations can not be supported by the TRS. In particular, those expressions containing the forms (indexing some rows) that are not related to any other forms and they are not normal forms.

As an illustration, let figure 7.2 represent a transitive closure matrix $M^+$. 
Assuming that the forms, $F_i, \ldots, F_j$, are not in their normal forms, this indicates an insufficiency in the set of rules (from which the relation is derived).

Although, the matrix does not identify explicitly the pairs of forms that are missing, it does supply the user with useful information. It identifies the un-transformed forms (in the row) and the range (in the column) from where the forms to pair with them (and possibly produce an ordered pair) are to be sought. This is an essential step forward in solving the incompleteness problem.

7.6 Space Considerations
The reason for our considering a sparse matrix is to minimize space usage, i.e. to involve a minimum space whenever it is possible.

7.6.1 Characteristics of Traditional "In-situ" Closure Algorithm
The closure algorithm increases the entries in the matrix $M$ in the process of producing $M'$. So, there may arise
situations when closure process produces large increases in matrix entries. In such cases the matrix may cease to be sparse. But if we consider the representation of the complement of the matrix, the closure process will cause the number of entries to be reduced and become more sparse rather than less. For example, in a closure matrix depicted in figure 7.3, more than 4/5 of it is full of entries (represented by the shaded part) and just under 1/5 is empty.

![Acceptable Forms](image)

Figure 7.3

It can be seen that less space is needed to hold the 1/5 than the 4/5. So, by transforming the closure matrix to that in figure 7.4 which represents its inverse, we obtain a sparse matrix and save a considerable space.
7.6.2 Complement Form

We foresaw the decline in efficiency that would ensue as more entries are involved in a closure matrix. To maintain this efficiency we thought of storing the information about the complement of the closure matrix when the latter consumes more than the necessary space. In order to achieve this we used a modified Warshall's algorithm which, instead of increasing the entries decreases $M^+$ to its complement (when necessary).

We should recall that the number of forms is known, this can be used to set a limit in the amount of storage allowed. If exceeded in the process of computing the closure matrix, the computation switches to the calculation of the complement. For convenience we set the limit to $1/2[n*n]$, where $n$ is the number of forms. When the number of the entries equals or is greater than the limit, the calculation of the closure is terminated and that of the complement is resumed. The address where the computation terminates is saved. The (whole) rows in the matrix are inverted, i.e. inserting
entries where there were none, and deleting the existed ones. For example, if row i at the time of the switch is:

```
Root
```

```
i 2
   ➔
   ➔
   ➔
i 4
   ➔
i 5
   ➔
i 9
   ➔
i 10
```

and the number of forms is 10, then the inverse will be:

```
Root
```

```
i 1
   ➔
   ➔
   ➔
i 3
   ➔
i 6
   ➔
i 7
   ➔
i 8
```

This is obtained by deleting the node_{j} s: \( j = 2, 4, 5, 9 \) and 10, and inserting node_{j} s: \( j = 1, 3, 6, 7 \) and 8.
The algorithm to achieve this, is:

**Algorithm 3:**

```
While row header address not equal to -1 do
  For i = 1 to n do
    If i = column number of an entry then
      Delete the entry
    Else
      Insert entry
```

After the execution of the above algorithm the computation of the complement is resumed. Here the modified Warshall's algorithm is applied. The outstanding difference between this algorithm and Warshall's is the application of the logical and as opposed to the logical or. Algorithm 4, below gives the modified procedure that computes the complement.

**Algorithm 4:**

```
While the address to the next column header (i) not equal to -1 do
  For i = 1 to n do
    If there is no node with index: [j, i] and j * i then
      Apply the logical and to the i\textsuperscript{th} row and j\textsuperscript{th} row, and replace j\textsuperscript{th} row by the result.
```
The application of the logical and involves:

Considering a counter from 1 to the number of the forms, if there is no entry at all in the \( i \)th row then delete all the entries in the \( j \)th row, if any. Otherwise, if the counter is equal to the column numbers of the nodes in the \( i \)th and \( j \)th rows, then do nothing. As long as the counter is equal to the column number of a node in the \( i \)th row but not to that in the \( j \)th row delete the node in the \( j \)th row. Notice that here deletion process is the valid operation unlike the insertion in the closure algorithm.

Let us illustrate the computation of the complement of a matrix in which the relation is encoded as 1s and non-relations as 0s. Assuming, after considering the 4th column of a matrix during the computation of the closure, the limit of the storage is exceeded, giving the matrix:

\[
\begin{bmatrix}
0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 1
\end{bmatrix}
\]

Notice that there are 16, 1s in the matrix which is greater than 13 \((= 1/2[5 \times 5] \) – rounding it to the next integer number). To resume the computation of the complement, the address will be incremented – in this case we will consider the last column (i.e. the 5th column). But, before that, we
must calculate the inverses of the rows, doing so we obtain the matrix:

\[
\begin{bmatrix}
1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

Now, by algorithm 4, we increment the address of the column header - which means pointing to column 5. Under the column, in the first row, the entry is 0 indicating there is no node and 1\#5, obviously the requirement is satisfied, so, we apply the logical \textit{and} to the first row and the fifth row and replace the first row by the result. The first entries are 1s in both rows, so we do nothing to the entry in the first row. The second entry is a 1 in the first row and a 0 in the fifth row, so the 1 in the first row is deleted and replaced by a 0. As we can notice, the remaining 0s stand, because (0 \textit{and} 0) is equal to 0. Proceeding in this manner, the second row will remain unchanged, the third and the fourth will change according. In particular, the 1 in the 3\textsuperscript{rd} column of the third row is replaced by a 0, whereas, the 1s in the 2\textsuperscript{nd} and 3\textsuperscript{rd} columns of the fourth row are also replaced by 0s.
On termination of the algorithm, the complement matrix:

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

is obtained.

To prove that it is the complement of the closure matrix, let us go back and complete the computation of the closure matrix (which was suspended when the storage limit was exceeded). Remember that the computation terminated after the 4th column, so, we need to consider just the last column. Again, remember that in the algorithm we look for 1s (i.e. node$_{j1}$ and j≠i). Oring the first and fifth rows changes the first, by replacing the second 0 entry to 1. Similarly, the third and fourth rows will change to produce the closure matrix:

\[
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\]

It is now obvious that the last but one matrix is the complement of the above closure matrix. Notice that storing the 1s is more expensive (space-wise) than storing the 0s in the closure matrix, and the 0s actually represent the
complement. The importance of this alternative representation becomes very indispensable when we think of a very large number of forms. So, we have in this way maintained the profound efficiency in the space management.

We have demonstrated the calculation of the complement matrix and how it successfully helped in saving space. However, we have said nothing about how it can assist in the problem of sufficiency of the equations (rewrite rules). This is very simple. Assuming that:

\[
\begin{array}{c}
\text{Acceptable Forms} \\
\end{array}
\]

represents a closure matrix, then the matrix below:

\[
\begin{array}{c}
\text{Acceptable Forms} \\
\end{array}
\]

represents its complement. The vertical shaded part of the above matrix does not count since it is outside the range of the acceptable forms. We then concentrate on the two horizontal strips. These shaded parts represent sequences of
entries and they identify some forms at some rows; the set of
those cannot be transformed to acceptable forms. To get the
set of forms for which the set of rewrite rules is sufficient
is a simple arithmetic, i.e. subtracting the identified set
from $\Sigma$. Thus, any subset of $L(G)$ consisting, exclusively, of
the forms obtained the difference is guaranteed to be
normalized using the set of rewrite rules. We have shown
that in either way, the final matrix supplies the same
information about the set of rules.

7.7 Conclusion
We have discussed how to determine whether or not a given set
of rules could support the transformation of every expression
in an algebra to a defined normal form. We have given a
matrix calculation for the transitive closure (and its
complement) of a relation obtained from a given specification
of an algebraic system. Both the closure and complement
matrices represented the same information about rules, but
where the former needed a lot of space the latter was the
better alternative. Also, the both calculations supplied
guidance to the user on how he could go about completing an
incomplete set of rules.
8. COMBINED ALGEBRAIC SYSTEMS

An algebraic system identifies a particular sort (of objects to be manipulated) and related operations. Usually, a combination of such systems involves a mixture of sorts, and this brings about gaps in the system when viewed as a whole. In particular, we cannot have a homogeneous system if there are no operations linking the sub-systems. The same idea of homogenization was employed by Yelick, [Yel 87], in his unification of combined systems. The difference is that he considered individual terms, whilst we intend to consider entire algebraic (sub-) systems. Nevertheless, there are natural partitions existing in composite system which follow the 'types' occurring in the signatures of the operators.

In the first section of this chapter we discuss the introduction of new operators as an example of how to bridge the distinction between different systems. The following section considers the combined TRS, i.e. the combination of the individual TRSs. We also discuss how, given a large complex system, the matrix calculations described in chapter 7 may be used in obtaining a suitable hierarchy of sub-systems in which our normalization methodology can usefully be applied.

8. Introduction of New Operators

As mentioned earlier, a combination of algebraic systems gives a mixture of sorts (and, consequently, operations), which in most cases are quite distinct. The composite system
needs to be homogenized before any expression consisting of the different sorts and operations can be formulated and subsequently rewritten. This entails defining operations which link the systems. The new operation must be capable of delivering the sort associated with one algebra when applied to operands derived from another. This is the only way that the operation can bridge the gap between the systems.

For example, let us consider the composite system consisting of a Boolean algebra and the ring of integers. Obviously they have distinct sorts: bool (which we denote by $\mathcal{B}$) and int (denoted by $\mathbb{Z}$) and operator sets: \{false, true, $\neg$, $\vee$, $\wedge$\} and \{0, 1, +, *\} respectively (of course $\mathbb{Z}$ has an infinite set of constants, but only 0 and 1 are used within the axioms). A homogeneous system is obtained if some of the relational operators: \{=, $\neq$, $<$, $\leq$, $>$, $\geq$\} are introduced. These operations when applied to integers produce values in $\mathcal{B}$. For example, the relational operator symbol: $<$, when applied to the integers $n$ and $n+1$ delivers the logical result true, i.e.

\[ n < n+1 = \text{true} \]

The operator '$<$' has signature:

\[ \mathbb{Z} \times \mathbb{Z} \rightarrow \mathcal{B} \]

Also, in terms of the sets of operators, there is a partition depicted in figure 8.1.
The interpretation here is simply that, in an expression (tree) from the composite system, the boolean operators are nearer to the root; next come the relational operators and finally the integer operator symbols. A typical example is given in the tree in figure 8.2.

Here \( z_1 \) and \( z_2 \) are integer variables. Note the partition on the sets of the operator symbols in the figure. The signature:

\[
\mathbb{Z} \times \mathbb{Z} \to \mathbb{B}
\]

itself constitutes a hierarchy on sorts. No logical (boolean) expression can produce an integer result, thus
indicating the "superiority" of $\mathbb{B}$ over $\mathbb{Z}$ in the hierarchy. This can be represented in a tree structure as in figure 8.3.

![Figure 8.3](image)

The 'rel' in figure 8.3 denotes any relational operator symbol. We know that we have signature such as:

$$\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$$

but there is no mixture of sorts of the kind:

$$\mathbb{Z} \times \mathbb{B} \rightarrow \mathbb{B}.$$  

In most cases, the hierarchy on the sorts may be derived from the signature(s) of the sorts of the constituting algebras. That is, for any two arbitrary sorts $A$ and $A'$, if there are signatures

$$A \times A \rightarrow A'$$

but not

$$A' \times A' \rightarrow A,$$

then, the hierarchy on the sorts of the algebras is

$$A' \uparrow A.$$  

8.2 Combined Rewriting Systems

The combined system will include all the rewrite rules of the individual systems together with those to deal with the
linking operators. In particular, in the case of boolean and integer combined system, the rule:

$$x < y \rightarrow \text{false} \quad \text{if} \quad ((x = y + z) \land z > 0)$$

may be added to the rule set.

In the normalization of the relational expressions it is essential to remove un-necessary operators which would otherwise cause some ambiguities. This may take the form of restricting the number of the relational symbols and then using negation in conjunction with the restricted symbols to form others. For example, if we consider only the set \{=, <, \leq\}, to get the set \{\#, >, \geq\}, we can write \(\neg(=)\) in the place of \#, \(\neg(<) \land \neg(=)\) for > and \(\neg(<)\) to mean \(\geq\).

Illustrating this with an expression say \(x > y\), this is then written as \(\neg(x < y) \land \neg(x = y)\).

We can then remove operators from the set \{=, <, \leq\} since

\[
\begin{align*}
x \leq y & \equiv (x = y) \land (x < y), \\
x < y & \equiv (x \leq y) \land \neg(x = y)
\end{align*}
\]

and

\[
\begin{align*}
x = y & \equiv (x \leq y) \land (y \leq x).
\end{align*}
\]

So far the set of 6 relational operators have been reduced by 'mixing' them with operators from the Boolean algebra. Depending on what operators are available within the lower (arithmetic) sub-expressions, we might be able to further reduce the forms that we need to include in the normalized expressions.
If subtraction is introduced via
\[ x - y = z \triangleq x = y + z, \]
then \( y < x \) may be transformed into
\[ 0 < x - y. \]

Similarly,
\[ x = y \triangleq 0 = y - x. \]

This then reduces the normalized form of \(<\) and \(=\) to
\( \{0\} \times Z \to B \) from \( Z \times Z \to B \), and removes the ambiguity that arises from being able to 'spread' the expression across the left and right hand sides of a relation.

When appropriate division can be removed by using relations such as:
\[ x < y/z \triangleq (x \ast z < y) \land (z \neq 0) \]
and hence
\[ 0 < y - (x \ast z) \land \neg (0 = z). \]

Notice that figures 8.1 and 8.3, also represent the trees indicating the levels where rewritings in the respective algebras are valid. It is therefore more profitable at each stage of reduction only to consider the rules applicable to the appropriate sub-system, not to the whole system.

The term-rewriting strategy we presented in chapter 4, (i.e. layered system), fits perfectly with hierarchical composite systems. The application of the rewrite rules associated with the appropriate part at each stage allows us to handle the overall system more efficiently. Furthermore, the methodology will improve the rule searching time because
unnecessary rules will not be scanned. The effect of this is immense when a significantly large number of rules are involved. In summary, the strategy reduces a complex problem into a progression of simpler ones.

8.3 Matrix Calculations

Here we index the matrix with the forms which include type checks. When the matrix calculations of chapter 7 are performed on a combined system, it not only addresses the completeness problem, but also gives assistance in the partitioning of rewriting rules. The differences in the sorts and operations of the composite system are highlighted by the "divisions" in the final matrix. As in chapter 7, the entries of the matrix will identify the non-normalizable forms if the set of rules is not sufficient. But in addition there are partitions reflecting the difference in the sorts and operations of the composite system.

The algorithm of chapter 7 that computes the forms (combinations of operator symbols) must now be modified to include the type checks mentioned earlier. In a simple system, where only one sort is involved, there is no need for a formal check on the sorts of the operator symbols. However, this is different for combined systems. In a system consisting of Boolean and integer systems we cannot make use of forms like:

\[ \lor (\neg, *) , \neg (+) \text{ and } * (\lor, \land) . \]
A standard form that can be sought by a combined system, may be a composition of the standard forms sought after by the component systems. That is, for any expression from a combined system, the standard tree will show the standard forms of each algebraic system at its individual level in the tree, and the normalization of the relational operations. For example, the tree given in figure 8.3 shows the boolean algebra at the top, the integer below and the relational operations in the middle. Therefore, a standard form for the combined system may be defined as:

\[
\begin{align*}
B &::= B_1 \mid B \lor B_1 \\
B_1 &::= B_2 \mid B_1 \land B_2 \\
B_2 &::= B_3 \mid \neg B_3 \\
B_3 &::= R \mid B_{\text{var}} \mid B_{\text{const}} \\
R &::= Z = Z \mid Z < Z \\
Z &::= Z_1 \mid Z + Z_1 \\
Z_1 &::= Z_2 \mid Z_1 \ast Z_2 \\
Z_2 &::= \text{mult}.Z_{\text{var}} \mid Z_{\text{var}} \mid Z_{\text{const}} \\
\text{mult} &::= Z_{\text{const}}
\end{align*}
\]

Notice that the first part of the definition is exactly the same as the intermediate standard form considered for the boolean expressions in chapter 4. This may then be linked by normalized relational operations to integer expressions in standard form.
8.4 Conclusion

In this chapter we have shown that, by their very nature, composite systems may contain incompatibilities of sorts and operations. The introduction of appropriate new operation(s) allows the production of a homogeneous system, the terms of which can subsequently be transformed in stages. The order they could be best exploited depends on the signature(s) of the operators contained. Matrix calculations on a combined system can be used to facilitate the hierarchical decomposition into sub-systems.
9. APPLICATIONS OF TERM-REWRITING

Apart from the attempt to mechanize theorem proving by term-rewriting very little work on term-rewriting applications is of sufficient maturity to have reached the general literature. Nevertheless it is widely regarded as having great potential, particularly with the advent of logic and functional programming languages the behaviours of which act in accord with traditional mathematics. We present some areas of computing where term-rewriting has been successfully applied, and also we mention one specific area where we believe that the strategy can be of substantial benefit.

The first section of this chapter, discusses the use of term-rewriting as an execution strategy for certain programming languages. The next section identifies its use as a program verification tool. Lastly, section 9.3 outlines its likely exploitation in the field of program transformation for the purpose of optimization.

9.1 In Specification Languages

9.1.1 The OBJ Language

OBJ is a language for writing and executing abstract formal specifications of programs. It is based on an algebraic model of computation rather than on the more usual logical models. It has an operational semantics given by rewrite rules (see [Gog 79] and [WGCS 1.0]).
The denotational semantics of OBJ is algebraic, and follows the style used to define Abstract Data Types axiomatically. In particular, the denotation of an object is an algebra which is initial in the category whose objects are models satisfying the equations and whose morphisms are maps between the respective universes and which commute with the interpretations of each operator. A universe for the initial model can be obtained by first constructing the set of ground terms, i.e. the Herbrand universe, for the given operators. The equations generate an equivalence relation on these terms and the corresponding quotient is the universe of the initial algebra. The operators are interpreted in the usual way as functions with arguments in this universe chosen to respect the signature and operate in the expected manner.

Initiability is important for several reasons:

1. The initial algebra, as semantics of a specification, is characterized uniquely up to isomorphism by the simple property that there is a unique homomorphism into each algebra of this specification.

2. The initial algebra reflects many of the properties that are desirable for semantics of an algebraic specification. A specification is a statement concerning the existence of data, operations and properties. The semantics of a specification should not add to this statement on an uncontrolled way. The initial algebra of an equational specification is an ideal example of such "minimality" with respect to the existence of data, operations and properties. Since it is generated it contains no data which are out of
reach of the operations (no junk), in other words, it contains only those data values that the specification allows you to speak about. As an algebra to the specification and indeed an (abstract) realization (of the specification) it does not have ground equations that are not necessarily true in any realization of the specification (no confusion).

The completeness property of the initial model asserts that any two ground terms which have equal value in the initial algebra can be proved equal by equational reasoning.

The adoption of initial algebra semantics for OBJ specifications causes the equational logic to become incomplete for the associated equational theory when variables (i.e. non-ground terms) are considered. The restriction to a specific model means that a structural induction principle must be used to establish most properties of a specification which involve non-ground terms.

The operational component of OBJ is based on rewrite rules. These are written as equations, but are interpreted as ordered pairs:

left-hand-side → right-hand-side

and are used as rules for replacing one term by another.
For example (cf. [Gog 84]), the equation
\[
\text{Pop(Push(Item, Stack)) = Stack}
\]
interpreted as rewrite rule, is used to rewrite the term:
\[
\text{Top(Pop(Push(2,Push(1,Empty))))}
\]
to the term:
\[
\text{Top(Push(1,Empty))}
\]
by replacing
\[
\text{Pop(Push(2,Push(1,Empty)))}
\]
by
\[
\text{Push(1,Empty)}
\]
Here Item and Stack are variables of sort integer and stack respectively.

Under a certain constraint, called finite convergence, the operational semantics assigns a unique canonical term to each equivalence class of ground terms in the initial algebra. Term-rewriting provides an effective decision procedure for evaluating terms in an equational theory presented via a set of (conditional) equations.

In OBJ the operator symbol: '==' checks to see if its two arguments are in the same equivalence class by rewriting each to its canonical form and then testing syntactically that the two forms are identical.

The unique convergence is based on the
- Church-Rosser property (C-R property)
- unique termination property.
Let $X$ be a set, and $\mathcal{R}$ be a relation on $X$, so $\mathcal{R} \subseteq X \times X$. Think of $(x, y) \in \mathcal{R}$, as meaning that $x$ can be rewritten to $y$. Now let $\mathcal{R}^*$ and $\mathcal{R}^\dagger$ denote reflexive-transitive and reflexive-transitive-symmetric closures of $\mathcal{R}$. Then $\mathcal{R}$ has the C-R property if and only if $(x, y) \in \mathcal{R}^*$ and $(x, y') \in \mathcal{R}^\dagger$ implies that there exists a $z \in X$ such that $(y, z) \in \mathcal{R}^*$ and $(y', z) \in \mathcal{R}^\dagger$.

$y \in X$ is $\mathcal{R}$-reduced if and only if there is no $z \neq y$ such that $(y, z) \in \mathcal{R}^*$; $y \in X$ is $\mathcal{R}$-reduced form of $x \in X$ if and only if $(x, y) \in \mathcal{R}^*$ and $y$ is $\mathcal{R}$-reduced. Notice that $\mathcal{R}^\dagger$ is an equivalence relation, so it make sense to talk of $\mathcal{R}^\dagger$-equivalence classes. Then $\mathcal{R}$ has a unique termination property if and only if $\mathcal{R}^\dagger$-equivalence classes contains at most one reduced form.

**Proposition** (cf. [Gog 79]) If $\mathcal{R}$ is C-R, then $(x, y) \in \mathcal{R}^\dagger$ and $y$ is $\mathcal{R}$-reduced implies $(x, y) \in \mathcal{R}^*$. 

The equations defining the operations in an OBJ program must satisfy both these two properties. In practice, it is straightforward to produce a set of rewrite rules with the necessary properties. The C-R property can be achieved by choosing a set of equations in which one lhs can ever match a ground term (with optional condition holding).

### 9.1.2 Realization of Function Definitions

In a logic programming language with equality the language is made of equational clauses, i.e. Horn clauses (clauses that have at most one positive literal) where the only predicate
is equality. Other predicates $P$ are viewed as Boolean functions and are of the form $P=true$. A program is a finite set of equational clauses. The computation process of the language is not based on resolution as in the case of PROLOG, but on clausal superposition. Clausal superposition (overlapping), roughly speaking, is an oriented form of replacement of "equals by equals". Thus, it is more time and space consuming than resolution, since unification applies not only to the goal literal but to all the subgoals of the literals. To cope with equational clauses, an interpreter must include important strategy refinements in order to be competitive with PROLOG.

In addition to the equational clauses, a specific set of rewrite rules is included within each program. Rewrite rules are inductive consequences of the program. The superposition process is followed by a rewriting process which simplifies subgoal formulae and thus reduces the storage space.

Fribourg (Fri 85), in his implementation of a logical interpreter of equational clauses, SLOG, used a rewriting strategy with clausal superposition. Superposition was originally used by Knuth and Bendix [KnBe 70] in their calculation of critical pairs in their completion procedure. The process, as described in chapter 2, is applied to two oriented equations (rewrite rules). The clausal superposition is an extension of superposition to handle conditional equations which can then be used as a computation rule.
The purpose of utilizing rewrite rules is two-fold:

1. to implement eager evaluation, and
2. as a means of detecting failure.

Rewriting provides eager evaluation by performing several superpositions in one step. In contrast with superposition, rewriting replaces goals by their "final" forms and does not store the intermediate rewrite-forms, thus gaining memory space and reducing the search tree.

Rewriting is a means of detecting failure since any rewriting of a subgoal:

\[ M = \text{true} \text{ (or } M = \text{false}) \]

to the contradictory literal:

\[ \text{false} = \text{true} \text{ (or } \text{true} = \text{false}) \]

allows for the rejection of the whole goal.

9.1.3 Functional Parallel Programming

Functional Parallel Programming (or FP2) is a programming language which combines functional and parallel styles through a term-rewriting strategy [Sch 88].

In FP2, Data types are defined as term algebras, and functions over these types are defined by equations with an operational semantics, just as in OBJ.
An example is:

<table>
<thead>
<tr>
<th>Type</th>
<th>Nat</th>
</tr>
</thead>
<tbody>
<tr>
<td>cons</td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>→ Nat</td>
</tr>
<tr>
<td>S:</td>
<td>Nat → Nat</td>
</tr>
<tr>
<td>opns</td>
<td></td>
</tr>
<tr>
<td>*+,+</td>
<td>Nat,Nat → Nat</td>
</tr>
<tr>
<td>eqns</td>
<td></td>
</tr>
<tr>
<td>x+0</td>
<td>→ x</td>
</tr>
<tr>
<td>x+S(y)</td>
<td>→ S(x+y)</td>
</tr>
<tr>
<td>x*0</td>
<td>→ 0</td>
</tr>
<tr>
<td>x*S(y)</td>
<td>→ (x*y)+x</td>
</tr>
</tbody>
</table>

Nat is isomorphic to the set of natural numbers and the symbols 0 and S are the constructors.

The operational semantics of FP2 is based on non-deterministic rewriting. By combining processes it is possible to build parallel network of them. A simple example of a FP2 process may be a program that updates the state Q of a buffer which we give diagrammatically as:
This can be given by the rules:

\[ \rightarrow Q(\text{nil}) \quad \text{(representing an empty buffer)} \]

\[ Q(s) : I(x) \rightarrow Q(s.\langle x\rangle) \quad \text{\{the event input changes the state by adding a term to the buffer\}} \]

\[ Q(\langle x\rangle.s) : O(x) \rightarrow Q(s) \quad \text{\{the event output changes the state by taking out a term from the buffer\}} \]

A trace to show how the process works is as follows:

\[ Q() \rightarrow I(0) \quad Q(0) \rightarrow I(1) \quad Q(0.1) \rightarrow O(0) \quad Q(1) \rightarrow I(2) \quad Q(1.2). \]

9.2 **As a Program Verification Tool**

Term-rewriting can be employed as a proof tool in software validation. The SPADE Proof Checker [O'Ne 88], is a tool designed for use in the formal verification of computer programs written in imperative languages. These languages can be either high-level (e.g. PASCAL) or low-level (i.e. assembly code). The Proof Checker works in conjunction with other static analysis tools. These do not execute the code with actual values (as in case of program testing), but rather operate on a rigorous model of the program.

Given a model, a flow analysis tool is used to check that the program is well-formed in respect of

1. control-flow analysis,
2. data-flow analysis, and
3. semantic analysis.
1. Control-flow analysis:— When a program is written in a high level language such as PASCAL and it makes little use of labels, the control flow is fairly obvious from the text, and any flaws in the control structure (such as unreachable code) are often easy to find by inspection. In contrast, programming languages such as FORTRAN and BASIC (and especially in assembly languages) usually require us to make extensive use of labels. It is still possible to write well structured programs, but even with a well written program the control structure is not always obvious from the program listing. Control-flow errors are easily made (for instance by misspelling a label) and difficult to detect. In such circumstances it can be useful to check the control structure systematically.

Some accessibility tests given, by Carré [Car 80], are listed below. A program should not contain —

1. jumps to non-existing labels,
2. labels to which no jumps are ever made,
3. statements which can not be reached from the start statement, and
4. statement from which one can not reach a halt statement.

2. Data-flow analysis:— This is used in validation systems to detect program errors such as references to undefined variables. These are important because they are often indicative of common programming errors such as misspellings, confusion of names and omission of statements.
Let us consider references to undefined variables. Formally, a (re-) definition of a variable may be by a program statement which modifies its value. A path in a control-flow graph is definition clear for a variable if there is no definition of the variable along the path. Finally a re-definition over-rides the previous definition of a variable.

Definitions that reach each of the nodes of a program can be represented by a boolean vector (or matrix) with one entry for each variable definition of the program; this entry taking the value 1 if there is a definition of the variable and 0 otherwise. Using the reference statements of the variables in the program and cross checking with the vector it can be established whether any variables are referenced before they were actually defined.

3. Semantic analysis:— If the flow analysis of the program is satisfactory, then, the SPADE semantic analysis tools are used. These are concerned with checking the behaviour of a program at the semantic level, with the ultimate aim of showing that it meets its specification. Since the specification may be very complex, and is usually only available in informal terms, full verification necessarily involves a great deal of mental work on the part of the programmer. At this level software verification aids can only assist the programmer in his task by relieving him of the tedious, mechanical process involved in reasoning about a program.
In practice it is very difficult to prove the correctness of a program. In many software verification aids the basic concept involved is the use of assertions and path traversal conditions. There is also the related symbolic execution of the program, and this is where rewriting strategies play important role. By symbolic execution of a program we mean its execution with symbols as the input values, in the place of actual data objects, e.g. integer numbers.

One basic method of proving the partial correctness of a program is to attach assertions, i.e. precise statements of what are believed to be true about the program variables at that point. In particular, an assertion attached to an output node of a program may state that the values output at that point bears the required relationship to the input. Then it is left to prove that the truth of the assertion at each point follows from the truth of assertions at previous points in the program.

The choice of the nodes at which assertions are planted naturally decomposes a program into elementary paths. The final stage of program verification involves taking each such path in turn and checking the consistency between the assertions at its endpoints. Take a path, with initial endpoint $s$ and terminal endpoint $t$ say, and with assertion $A_s$ and $A_t$ at these points respectively. We must check that if $A_s$ is true when the program execution reaches point $s$, then $A_t$ will be true on reaching $t$. The simplest way of doing this is to "pull up" the assertion $A_t$ backwards up the path,
transforming it along the way by taking into account the effect of all the assignment statements and the outcome of all tests which are encountered on the path to \( s \). The result is a modified form of \( A_t \) and it is denoted by \( A_t' \). Clearly, if the program execution traverses the path \( s \) to \( t \), then \( A_t \) will be true at \( t \) if and only if \( A_t \) was true at \( s \). Hence the truth of \( A_s \) at \( s \) implies the truth of \( A_t \) at \( t \) if and only if

\[ A_s \supset A_t' \]

( where \( \supset \) is the implication symbol). The formulae (i.e. \( A_s \supset A_t' \) ) are kept in as simple a form as possible. Using, in particular, the rewrite rule, \( x \supset y \rightarrow \neg x \lor y \), the above implication can be replaced by its equivalent

\[ \neg A_s \lor A_t' \]

which is 'simpler'.

For instance, consider obtaining the formula,

\[ (A_s \supset A_t') \land (A_t \supset A_v') \]

from the path \( s \) through \( t \) to \( v \). By applying the rule, \( x \supset y \rightarrow \neg x \lor y \), twice the above expression will yield

\[ (\neg A_s \lor A_t') \land (\neg A_t \lor A_v') \]

Notice the similarity between this expression and the Boolean expressions we considered in chapter 4. Assuming the same normal form (as in chapter 4),
notice that the first stage of our system will transform the logical expression

\[ (\neg A_e \lor A_t') \land (\neg A_t \lor A_v') \]

to

\[ (\neg A_e \land \neg A_t) \lor (\neg A_e \land A_v') \lor (A_t' \land \neg A_t) \lor (A_t' \land A_v') \]

(The rules applied are: \( x \land (y \lor z) \rightarrow (x \land y) \lor (x \land z) \) and \( (x \lor y) \land z \rightarrow (x \land z) \lor (y \land z) \) respectively.)

The SPADE Proof Checker is a first-order classical logic proof tool, using a natural deduction system. The components of the Proof Checker that realize the algebraic simplification are:

- expression standardizer,
- simplifier, and
- external proof rules.

These components require rewriting processes. The reader is referred to [Car 80], [ClCa 88] and [O'Ne 88] for further discussion on these points.

9.3 Program Transformation

A new area in which we believe term-rewriting can play an important role is in formal transformation of programs.

In chapter 5, we described a general method for term-rewriting in the simplification of expressions, i.e. applying some directed equations to replace sub-expressions of the original expression until no more replacements are possible. Now, if we replace expressions by program schemes and the equations by transformation rules then there is no reason why
the same methodology cannot be adopted to obtain an optimized program from a given one. The only difference will be the criteria for orienting the transformation rules (which must be given as equations). Possible criteria may be time, space or other 'complexity' measures.

In much the same way as certain equations relate to particular algebras (as mentioned in chapter 7), it is likely that certain sets of transformation rules will relate to a particular "type" of program. After all, we would not dream of attempting to use equations for field to solve problems in group theory.

For a program model the set of transformation rules may be given as:

\[
\{ T_{li} = T_{ri}, \text{ for } i=1 \text{ to } n \}
\]

where \( T_{li} \) and \( T_{ri} \) represent the left and right hand sides of rule \( i \).

Suppose that for a given program segment \( P \) we can compute a suitable metric; denoting the result of applying the metric to the program schema \( P \) by \( M(P) \), we can say that the equation

\[
T_{li} = T_{ri}
\]

directs to the right only if \( M(T_{li}) > M(T_{ri}) \), where > is the natural order relation on the relevant metric space. The directed transformation rule is then given as:

\[
T_{li} \rightarrow T_{ri}.
\]
By replacing the match of $T_{l_1}$ in a program $P$, by $T_{r_1}$ achieves a rewrite, and the program $P[T_{l_1} \leftarrow T_{r_1}]$ is obtained. $P[T_{l_1} \leftarrow T_{r_1}]$ denotes the resulting program after the replacement of $T_{l_1}$ by $T_{r_1}$. When all possible rewrites has been achieved the derived program is "optimized".

Devising a metric measure is indeed not a trivial problem, it is an area of research on its own. However, let us illustrate a simple example of a transformation of a sequential system to its equivalent parallel system. The major problem involved is the identification of independent component parts of the sequential system and processing them in parallel. The procedure required to solve this problem is the same as the searching and replacement processes encountered in the TRS for expressions. That is the analysis and assignment of some parts of the sequential system for parallel processing.

Programs describe systematic procedures in structural programming languages. Two programs are considered equal (for this purpose) if for any input data both produce the same output.
Consider this simple sequential procedure $P$ given in flow chart in figure 9.1.

![Figure 9.1](image)

Notice that the first loop does not contribute to the outcome of the second, in other words they are independent. Without going into technical details, a transformation rule can be given pictorially as in figure 9.2.

![Figure 9.2](image)

Assuming that the measures for the two loops are $M(P_1)$ and $M(P_2)$ and $M(P_3)$ for the last part of the original computation. The two combinations can be represented by seq
and \( \text{par} \) and these will themselves give rise to formulae for the 'complexity' of these combinations, so let
\[
\mathcal{M}_\text{seq}(P_1, P_2) = \text{Seq}(\mathcal{M}(P_1), \mathcal{M}(P_2))
\]
and
\[
\mathcal{M}_\text{par}(P_1, P_2) = \text{Par}(\mathcal{M}(P_1), \mathcal{M}(P_2))
\]
where \( \text{Seq} \) and \( \text{Par} \) are functions that stipulate how the measures combine. If our metric was execution time then may be
\[
\text{Seq}(\mathcal{M}(P_1), \mathcal{M}(P_2)) = \mathcal{M}(P_1) + \mathcal{M}(P_2)
\]
and
\[
\text{Par}(\mathcal{M}(P_1), \mathcal{M}(P_2)) = \text{max}(\mathcal{M}(P_1), \mathcal{M}(P_2)) + r
\]
(where \( r \) represents 'rendezvous' time) would be reasonable definitions.

Since metrics (which would better be called norms, but the literature traditionally uses the wrong term) are non-negative, following the 'time' metric we have
\[
\text{Seq}(\mathcal{M}(P_1), \mathcal{M}(P_2)) > \text{Par}(\mathcal{M}(P_1), \mathcal{M}(P_2)).
\]
Therefore, the transformation rule orients to the right. To give the rewrite rule:

Figure 9.3
By replacing the match of the lhs of this rule in the program \( P \) by the rhs we obtain the flow chart given in figure 9.2 which is now optimized by processing the loops in parallel.

![Flow chart](image)

Figure 9.4

9.4 Conclusion

In the chapter we gave outline of the successful application of term-rewriting to some programming languages and program validation/verification. Also we draw attention to the need for a strategy in the application of program transformations, and leave the problem open for investigation.
10 CONCLUSION

We have presented a new methodology for term rewriting as used to normalize expressions in an arbitrary algebraic system, but yield significant benefits only when some operators are AC. The strategy involves the application of appropriate subsets of rules at certain stages within normalization. Due to this segmentation some rules are applied in different directions at different stages of the normalization.

From a set of algebraic identities and an ordering on the set of terminal symbols in the related grammar, we derive an ordering on the set of expressions allowed in the algebra. This ordering, called Hierarchical Operator Ordering (HOO), is then used to orientate the equations so as to obtain rewrite rules. The use of the precedence order on the symbols of the underlying grammar distinguishes HOO from other algebraically based orderings.

The initial and most fundamental stage in our normalization procedure may fail if the given set of equations is not sufficiently 'complete'. Although we were not directly concerned with completion procedures, we give a partial decision procedure by which the insufficiency of the original set of equations can be determined. An important characteristic of the procedure is that it identifies the areas in which normalization can fail.

The method can be extended to cope with composite systems providing that the associated 'types' can be layered
hierarchically. When appropriate, this results in the replacing of a complex normalization problem by a progression of simpler ones.

Our normalization process is most effective when the operator hierarchy contains enough AC operators to facilitate simplification at each layer. When this is not so, some of the stages of our TRS have to be omitted. In particular, if none of the operators is associative, much of the restructuring cannot be achieved. Although we could do without this restructuring, it means that the set of 'standard' expressions in the algebra is much larger. For example, consider the expressions in figure 10.1.

\[
\begin{array}{c}
\ast \\
 a \\
 \ast \\
 b \\
 \ast \\
 c \\
 \ast \\
 a \ast b \\
 \ast \\
 b \ast c \\
 \ast \\
 c \ast b
\end{array}
\]

Figure 10.1

Notice that they are syntactically different but contain the same symbols. However neither of them can be reduced further, so both have to be considered as being in standard form. Of course, introduction of *-associativity permits one of these expressions to be reduced to the other.

In addition to the obvious applications of term-rewriting in theorem proving, we gave brief accounts of some areas of computing where it can be put to good use. We also discussed the possible application of term-rewriting to the control of
program transformations. This is a non-trivial topic which promises a feast of interesting research problems. The example considered in chapter 9 gave a trivial in-sight of how the methodology may be used to guide program transformation. Perhaps this will give rise to practical techniques for 'tuning' correct programs into efficient ones and hence assist formal software engineering in reaching its much vaunted potential.

Finally we note that since our method yields deterministic expression reductions, it does not easily lend itself to parallel implementation. Parallel execution of processes is potentially of great value. Therefore an investigation of how the method might be modified to act (in parallel) on independent parts of an expression tree would also be of considerable practical interest.
REFERENCES


[Lan 75] Lankford, D. S., 'Canonical Algebraic Simplification in Computational Logic', Memo ATP-25, Automated Theorem Proving Project, University of Texas, Austin, TX, May 1975.

[Lan 77] Lankford, D. S., 'The Refutation Completeness of Permutation Narrowing and Block Resolution', ATP-35, Department of Mathematics and Computer Sciences, Austin, Texas.


