1982

Data structure implementation and correctness

Mahmoud Parsian
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
Parsian, Mahmoud, "Data structure implementation and correctness " (1982). Retrospective Theses and Dissertations. 8952.
https://lib.dr.iastate.edu/rtd/8952

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106
Parsian, Mahmoud

DATA STRUCTURE: IMPLEMENTATION AND CORRECTNESS

Iowa State University

University Microfilms International

300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy √
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Other

University Microfilms International
Data Structure
Implementation and Correctness

by

Mahmoud Parsian

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major: Computer Science

Approved:

Signature was redacted for privacy.
In Charge of Major Work
Signature was redacted for privacy.
For the Major Department
Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa
1982
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DEDICATION</strong></td>
<td>viii</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td><strong>1. INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1. Related Work and Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2. Outline of the Thesis</td>
<td>5</td>
</tr>
<tr>
<td><strong>2. PRELIMINARY BACKGROUND</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1. Basic Concepts and Notation</td>
<td>8</td>
</tr>
<tr>
<td>2.2. Context Free Grammar</td>
<td>9</td>
</tr>
<tr>
<td>2.3. Algebraic Syntax and Semantics</td>
<td>11</td>
</tr>
<tr>
<td>2.4. Derived and Represented Algebras</td>
<td>25</td>
</tr>
<tr>
<td><strong>3. REDUCTION ON DATA STRUCTURE TREES</strong></td>
<td>38</td>
</tr>
<tr>
<td>3.1. Algebraic Equations</td>
<td>38</td>
</tr>
<tr>
<td>3.2. Informal Description of a Rewrite Rule</td>
<td>43</td>
</tr>
<tr>
<td>3.3. Previous and Related Work</td>
<td>44</td>
</tr>
<tr>
<td>3.4. Application of a Rewrite Rule to a Tree</td>
<td>45</td>
</tr>
<tr>
<td>3.5. Reduced Trees under Rewriting Systems</td>
<td>51</td>
</tr>
<tr>
<td>3.6. A Lattice Theoretic Approach to Equivalences in Tree Rewriting Systems</td>
<td>61</td>
</tr>
<tr>
<td>3.6.1. Lattice theory background</td>
<td>61</td>
</tr>
<tr>
<td>3.6.2. Example: introducing L(ω)</td>
<td>63</td>
</tr>
<tr>
<td>3.6.3. Basic results about L(ω)</td>
<td>76</td>
</tr>
<tr>
<td>3.6.4. Algorithm for construction of L(ω)</td>
<td>79</td>
</tr>
<tr>
<td>3.6.5. Conditions for compatibility of L(ω)</td>
<td>85</td>
</tr>
<tr>
<td>3.7. Conclusions</td>
<td>88</td>
</tr>
</tbody>
</table>
4. NATURAL TREE TRANSDUCERS

4.1. Introduction to Tree Transducers

4.1.1. Construction of a tree transducer within $L(\theta_1)$ and $L(\theta_2)$

4.1.2. Informal construction of a tree transducer

4.1.3. Tree transducers, tree translation

4.2. Formal Definition of a Tree Transducer within $L(\theta_1)$ and $L(\theta_2)$

4.2.1. Algebraic definition of a tree transducer

4.2.2. Example: formal construction of a tree transducer

4.2.3. Tree transducer within $L(\theta_1)$ and $L(\theta_2)$

4.2.4. Mono tree transducer

4.3. Introduction to Properties of Tree Transducers

4.3.1. Source tree and transduced tree

4.3.2. Informal discussion of tree transduction

4.3.3. Informal discussion of consistency of tree transducers

4.4. Formal Definition of Consistency and Semi-consistency

4.4.1. Example: Boolean type implementation

4.4.2. Local consistency and local semi-consistency

4.4.3. Global consistency and global semi-consistency

4.5. Weak Syntactical Honest Tree Transducer

4.6. Major Theorems

4.7. Simplifications, Summary
5. CONDITIONAL REWRITING SYSTEM 122

5.1. The Idea of Conditional Rewrite Rules 122

5.2. Related Work and Discussion 122

5.3. Conditional Equations versus Simple Equations 123

5.4. Formal Definition of Conditional Equations 127

5.5. Examples: Power of Conditional Equations 132

5.5.1. Example 1: QUEUE 132

5.5.2. Example 2: BINARY TREE 136

5.6. Major Theorem 139

5.7. Summary and Results 142

6. DATA STRUCTURE IMPLEMENTATION AND CORRECTNESS 144

6.1. Related Work and Discussion 145

6.2. Algebraic Specification of a Data Structure 147

6.2.1. Definition of a data structure 147

6.2.2. Example of a data structure 148

6.3. Equivalence of Data Structures 149

6.3.1. Method for checking equivalent data structures 149

6.3.2. Example of equivalent data structures 150

6.3.3. Power of complex data structures 151

6.4. Implementation of a Data Structure 158

6.4.1. Syntactical level of implementations 159

6.4.2. Semantical level of implementations 161

6.5. Correctness of Implementation of Data Structures 162
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.6. Development of Correct Implementations</td>
<td>163</td>
</tr>
<tr>
<td>6.6.1. Development of SYNIMPL tree transducer</td>
<td>163</td>
</tr>
<tr>
<td>6.6.2. Correctness of SYNIMPL tree transducer</td>
<td>165</td>
</tr>
<tr>
<td>6.6.3. Development of SEMIMPL tree transducer</td>
<td>172</td>
</tr>
<tr>
<td>6.6.4. Correctness of SEMIMPL tree transducer</td>
<td>174</td>
</tr>
<tr>
<td>6.7. An Illustration of Correctness of Syntactical Implementation of Data Structures</td>
<td>176</td>
</tr>
<tr>
<td>6.8. Summary, Conclusions</td>
<td>189</td>
</tr>
<tr>
<td>7. CONCLUSIONS AND FUTURE WORK</td>
<td>190</td>
</tr>
<tr>
<td>7.1. Results and Summary</td>
<td>190</td>
</tr>
<tr>
<td>7.2. Directions for Future Research</td>
<td>193</td>
</tr>
<tr>
<td>8. BIBLIOGRAPHY</td>
<td>194</td>
</tr>
<tr>
<td>9. ACKNOWLEDGEMENTS</td>
<td>198</td>
</tr>
<tr>
<td>Figure 2.1.</td>
<td>Signature for &quot;set&quot; data structure</td>
</tr>
<tr>
<td>Figure 2.2.</td>
<td>The cfg for &quot;set&quot; data structure</td>
</tr>
<tr>
<td>Figure 2.3.</td>
<td>Commutative diagram: homomorphism</td>
</tr>
<tr>
<td>Figure 2.4.</td>
<td>Pictorial representation of $h&lt;f&lt;a&gt;h&lt;g&lt;b&gt;b&gt;$</td>
</tr>
<tr>
<td>Figure 2.5.</td>
<td>Syntax and semantics homomorphisms</td>
</tr>
<tr>
<td>Figure 2.6.</td>
<td>A row of a tree transducer</td>
</tr>
<tr>
<td>Figure 3.1.</td>
<td>Church-Rosser property</td>
</tr>
<tr>
<td>Figure 3.2.</td>
<td>Pictorial representation of $[[f]]$</td>
</tr>
<tr>
<td>Figure 3.3.</td>
<td>Pictorial representation of $[[t]]$</td>
</tr>
<tr>
<td>Figure 3.4.</td>
<td>Pictorial representation of Lemma 3.18</td>
</tr>
<tr>
<td>Figure 3.5.</td>
<td>$P({f_1, f_2, f_3})$ under binary relation $\geq$</td>
</tr>
<tr>
<td>Figure 3.6.</td>
<td>The lattice with four elements ($P(T)$)</td>
</tr>
<tr>
<td>Figure 3.7.</td>
<td>Effective lattice: $L(T)$</td>
</tr>
<tr>
<td>Figure 3.8.</td>
<td>Algorithm for computing $\pi_1 \Theta \pi_2$</td>
</tr>
<tr>
<td>Figure 3.9.</td>
<td>Algorithm for computing $\pi_1 \Theta \pi_2$</td>
</tr>
<tr>
<td>Figure 3.10</td>
<td>Unary and binary refinement function</td>
</tr>
<tr>
<td>Figure 3.11</td>
<td>All refinements function</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>$L(\phi_1)$ and $L(\phi_2)$</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Picture of an informal tree transducer</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Pictorial representation of tree transducer $T_2$</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Figure 4.4.</td>
<td>A tree transduction by tree transducer $T_2$</td>
</tr>
<tr>
<td>Figure 4.5.</td>
<td>Informal view of a tree transducer</td>
</tr>
<tr>
<td>Figure 4.6.</td>
<td>Full specification of tree transducer $T_1$</td>
</tr>
<tr>
<td>Figure 4.7.</td>
<td>Pictorial representation of $L(\omega_1)$</td>
</tr>
<tr>
<td>Figure 4.8.</td>
<td>Tree transducer $T$ within $L(\omega_1)$ and $L(\omega_2)$</td>
</tr>
<tr>
<td>Figure 5.1.</td>
<td>Equations for SET data structure</td>
</tr>
<tr>
<td>Figure 5.2.</td>
<td>Signature of QUEUE data structure</td>
</tr>
<tr>
<td>Figure 5.3.</td>
<td>Equations of QUEUE data structure</td>
</tr>
<tr>
<td>Figure 5.4.</td>
<td>Equations CE of BTREE data structure</td>
</tr>
<tr>
<td>Figure 5.5.</td>
<td>Representation of a binary tree (sort tree)</td>
</tr>
<tr>
<td>Figure 6.1.</td>
<td>Equations $E_1$, $E_2$</td>
</tr>
<tr>
<td>Figure 6.2.</td>
<td>Conditional and simple equations</td>
</tr>
<tr>
<td>Figure 6.3.</td>
<td>Pictorial representation of $R(0,m+1)$</td>
</tr>
<tr>
<td>Figure 6.4.</td>
<td>Pictorial representation of Theorem 6.19</td>
</tr>
<tr>
<td>Figure 6.5.</td>
<td>Specification of DEQUEUE data structure</td>
</tr>
<tr>
<td>Figure 6.6.</td>
<td>Equational specification of DEQUEUE data structure</td>
</tr>
<tr>
<td>Figure 6.7.</td>
<td>Specification of ARRAY data structure</td>
</tr>
<tr>
<td>Figure 6.8.</td>
<td>Equational specification of ARRAY data structure</td>
</tr>
<tr>
<td>Figure 6.9.</td>
<td>Implementation tree transducer $T$</td>
</tr>
<tr>
<td>Figure 6.10.</td>
<td>Dequeue representation</td>
</tr>
</tbody>
</table>
DEDICATION

In the Name of God
and
In the Name of the Heroic People of Iran
and
In the memory of the Great Mojahed, Moussa Khiabani, one of the most prominent leaders of the struggles for freedom in the contemporary Iranian history.

This dissertation is dedicated to

Mojahed Brother Massud Rajavi, leader of the People's Mojahedin Organization of Iran and in charge of the NCR.

The People's Mojahedin Organization of Iran (PMOI) and the martyrs of the PMOI and especially the martyred founders of the PMOI (Mohammad Hanif-nezhad, Said Mohsen, Ali-asghar Badizadegan) who shed their blood in the path of Emancipation.

---

1 The People's Mojahedin Organization of Iran (PMOI) was founded in 1965. Mojahedin believe in Islam as an ideology which can lead in abolishing of all kinds of dictatorship, colonialism and exploitation of man by man. The PMOI was the most active force in the struggle against the Shah's regime and after the February Revolution which caused the downfall of the Shah's regime, the Mojahedin started their new phase of struggle by fighting the suppressive and yet deceitful medieval theocracy of Khomeini.

2 The National Council of Resistance (NCR) is the united front of all the political forces and personalities loyal to Iran's "Independence
and Freedom", but opposed to the Shah's dictatorship and Khomeini's tyranny.

The Council was formed in July, 1981, and is the only popular and progressive alternative to the Khomeini's barbaric regime, enjoying an extensive support both internally and internationally.

The People's Mojahedin Organization of Iran is considered the main and most popular force within the council, while Mojahed Brother Massoud Rajavi is in charge of the NCR.
سام خدا
سام خلق حضرت ایران
و پدربزرگ شریف خلق حضرت ابراهیم حسین
و پدربزرگ بزرگ خلق حضرت ابراهیم حسین
رئیس‌الامام، سخن‌گویی‌های خود را در مورد حضرت ابراهیم حسین و پدربزرگش روی داده‌اند.

البته روایات زبان‌های بسیاری درباره حضرت ابراهیم حسین و پدربزرگش به‌وجود آمده است.

سالاریت
1. INTRODUCTION

Establishing correct implementation of data structures and correct translations from one language into another are important problems in computer science. The research reported here is primarily concerned with the definition and implementation of data structures, and how to prove an implementation correct.

Data structures [10, 11, 18] are needed in the design of algorithms to solve many problems. The practical application of data structures is an area upon which much attention is being focused. Indeed, specification and implementation of data structures is the basis for data base systems and software engineering. Currently algebraic concepts for specification of data structures [8, 9] are available. This thesis addresses itself to the determination of an adequate notion for the implementation of data structures.

Recent works on data structures suggest that the algebraic [8, 9, 10] or axiomatic [17, 18] approaches are useful for the specification of data structures. We believe that algebraic techniques are promising and appropriate for the specification of data structures, because concise and rigorous proofs can be outlined. Our specification of data structures is considered to be abstract, that is, implementation independent. The fundamental hypothesis of this approach is that the requirements to be met are specified by a set of equations and a semantic algebra, without referring to a real implementation.
The algebraic approach treats a data structure as an algebra with specifications giving properties in terms of equations that the operations of that algebra must satisfy. Basically, we will consider a data structure as a set of values together with a set of primitive operations and equations on those values.

This thesis deals primarily with developing criteria for providing provably correct implementation of data structures. To this end, we need to first develop a methodology to write equational specifications for data structures. That is, we design a specification for a data structure. One of the main reasons for designing a specification of a data structure is to have an implementation independent of any specific programming language. Thus, in our work, a distinction is made between a specification and implementation of data structures.

The work reported here is concerned mainly with formalizing the data structure implementation via "consistent" tree transducers. A tree transducer [3, 7, 28] is a mechanism for performing language translation and is used by us to determine the implementation of a source data structure by a target data structure.

To specify a data structure implementation, it is sufficient to specify the behavior of each operation associated with that data structure in terms of other (old) data structures that already have been implemented. Our approach, reduces the correctness of data structure implementation to the correctness of translation.
The syntactic information about a data structure is conveyed by its "signature", an algebraic machinery which gives the names of operations and domains, and specifies the domains and codomains of the operations. Semantic information associated with a data structure is conveyed by an algebra having the same signature. The properties of the operations are given by a set of algebraic equations. Algebraic equations (we develop two types, unconditional and conditional) enable us not only to specify the properties of data structures but also to check the equivalence and correctness of implementation of data structures.

Using the concept of equations on tree transducers, implementation of data structures will be defined on syntactical and semantical levels. Syntactical implementation of a source data structure by a target data structure is a mapping from equivalent class of elements of source data structure into equivalent class of elements of target data structure. In contrast, semantical implementation means mapping of elements of a source data structure into elements of a target data structure which have the same meaning.

Our work has been influenced by Guttag [13] and Goguen [8] for their work on the algebraic approach of data structures; by Engelfriet [7] and Krishnaswamy and Strawn [31] for their work on tree transducers.

1.1. Related Work and Motivation

There are two main approaches for specifying data structures, the abstract equational approach and the axiomatic approach. Liskov and
Zilles [34] give a survey of different specifications for data structures.

Algebraic specification techniques have been studied by several researchers including Guttag [13], Goguen [8, 9], Goguen, Thatcher, Wagner, and Wright [10] to mention a few. The algebraic approach to data structures has stimulated a number of languages for data structure specifications, including CLU [33] and ADA [44]. The axiomatic approach of data structuring, mostly, has been treated by Hoare [17, 18, 19].

Previous work [8, 9, 10] to address the issue of data structure implementations by used homomorphisms from one algebra into other. Our approach to this problem is by developing a tree transducer that specifies the implementation. One advantage of this is that by expressing data structure implementation as a translation problem, correctness of data structure implementation might be reduced to the correctness of translation.

This paper addresses the issue of data structure implementation via translation. That is, we associate each basic operation and composite operation of the source data structure by composite operations of the target data structure. Our implementation will be done by tree transducers.

Tree transducers have been examined by several researchers, including Baker [3], Engelfriet [7], and Thatcher [40] to mention a few. Our development follows that of Krishnaswamy and Strawn [31],
who investigated conditions to be imposed on tree transducers to guarantee that they induce semantic-preserving translations.

Goguen [8] addresses the issue of "errors" in abstract data types. In our work, this issue has been resolved by introducing conditional equations. Error values are associated with exceptional conditions, such as the underflow of a stack. In our work, it turned out that conditional equations enable us in specification of complex data structures.

The work developed in this thesis, approaches the implementation of data structures specified in a general way using conditional equations. Correctness of data structure implementation is examined at the "syntactic" and "semantic" levels. Syntactical correctness is associated with source and target equations; and semantical correctness is based on semantic homomorphisms.

We develop a lattice theoretic approach to derive the criteria for a correct implementation of data structures.

1.2. Outline of the Thesis

The second chapter introduces a formalism for defining tree transducers and specification of data structures. We first discuss essential facts about context-free grammars. Then we define a signature associated with a context-free grammar, which plays an important role in the specification of data structures. Later, in the algebraic framework, we define the concept of a language definition system, a pair comprising the syntax and semantic definition of a language.
The third chapter discusses algebraic equations, which are used in simplifying data structure elements. The algebraic equations define the behavior of a data structure. We view equations as rewrite rules, and then we define a lattice theoretic approach to equivalences in tree rewriting systems. This will later be used in development of "natural" tree transducers.

The fourth chapter discusses natural tree transducers. We discuss how the lattice theoretic development can be used as a basis for defining tree transducers. We first introduce tree transducers and then discuss certain properties of tree transducers, namely: "consistency" and "semiconsistency". The consistency and semiconsistency are syntactical properties of tree transducers using algebraic equations.

The fifth chapter extends the formalism of algebraic equations (Chapter 3) to conditional equations, a more powerful class of equations than the simple algebraic equations, considered in Chapter 3. Conditional equations enable us to define more complex data structures.

The sixth chapter presents specification, and correctness of data structure implementations. We identify three classes of data structure specifications. A method is stated for checking the equivalence of data structures. This is followed by the basic definition of the implementation of a source data structure by a target data structure. Two levels of implementations are identified: syntactical and semantical. We state basic definitions for the correctness of
of an implementation, and develop a methodology for proving the correctness of an implementation. This chapter concludes with a complete example illustrating the process of provably correct data structure implementation.

The seventh chapter contains our conclusions, and indications of directions for future research.
2. PRELIMINARY BACKGROUND

This chapter provides the basic terms and machinery needed to develop general mechanisms for "algebraic tree transducers", "consistent tree transducers", "semi-consistent tree transducers", and the "implementation of data structures". We develop an algebraic model by defining syntax and semantics, and later add "semantic equations" to this model. The rationale for this algebraic approach is to treat key issues in a concise and general manner, and also to be able to prove results rigorously.

The study of a language includes syntax and semantics. We will consider a language as a set of strings over an alphabet along with their meanings. Syntax of a language (syntax of strings) is specified using the mechanism of context-free grammars (cfg), and semantics of a language (meaning of strings) is specified using a mechanism similar to Knuthian synthesized attributes [25]. Before beginning our development, we present some notation and basic definitions which will be used throughout this paper.

2.1. Basic Concepts and Notation

If S is a set, then \(|S|\) is the number of elements of S, the empty set is denoted by \(\emptyset\), and \(P(S)\) is the set of all nonempty subsets of S. If A and B are sets, then \(A^B\) is the set of all functions from B into A. The term "tree" means a node-labeled ordered tree,
as defined by Knuth [26]. The height of a tree \( t \), denoted \( H(t) \).
If \( t \) consists of a single node, then \( H(t) = 1 \), if \( t = f(t_1, \ldots, t_n) \)
where \( f \) is the root of \( t \), then \( H(t) = 1 + \max \{ H(t_1), \ldots, H(t_n) \} \).
If \( S \) is a set and \( n > 0 \), then \( S^n \) is the set of all strings over \( S \) of
length \( n \). \( S^+ = \bigcup_{n>0} S^n \) is the set of all nonempty strings over \( S \),
and \( S^* = S^+ \cup \{ \lambda \} \) where \( \lambda \) denotes the empty string. If \( s \in S^+ \), then
\( s \in S^n \) for some \( n > 0 \), and length \( (s) = n \). If \( s = \lambda \) then length \( (s) = 0 \). An "equivalence relation" is a relation on a set that is re­
flexive, symmetric, and transitive. A "partition" of set \( S \) is a set
\( \{ B_1, \ldots, B_n \} \) such that
(a) \( B_i \neq \emptyset \) for \( 1 \leq i \leq n \);
(b) \( B_i \cap B_j = \emptyset \) for \( i \neq j \leq n \); and
(c) \( B_1 \cup B_2 \cup \ldots \cup B_n = S \)
The symbol \( \square \) is used to mark the end of theorems, definitions, examples,
and so forth.

2.2. Context Free Grammar

The following description of "context free grammars" is a quick
review of the development presented by Hopcroft and Ullman [20].
A "context-free grammar" (cfg) \( G \) is a quadruple \( G = (N, \Delta, P, Z) \)
where (1) \( N \) is a finite set of nonterminal symbols (usually denoted
by capital letters), (2) \( \Delta \) is a finite set of terminal symbols.
(usually denoted by lower case letters), (3) P is a finite set of productions, and (4) Z ∈ N is the start symbol. Furthermore, we assume that V = N ∪ Λ, Λ ∩ N = ∅ (the empty set), and P is a finite set of productions (rules) of which each element has the form A → w where A ∈ N and w ∈ V* (V* is the set of all finite words over V).

The set P induces a binary derivation relation ⇒ on V* defined as follows: if a, g ∈ V*, then a ⇒ g if and only if a = W1AW2, β = W1WW2, and A → w is a rule in P. Relation *⇒ denotes the reflexive and transitive closure of ⇒, i.e., α *⇒ β if and only if either α = β, or there exists φ1, ..., φm ∈ V* such that φ1 = α, φm = β and φi *⇒ φi+1 for all i (1 ≤ i < m).

The (set of) sentences of context-free grammar G, denoted by SEN(G) is the set \{w / w ∈ A*, and Z *⇒ w\}. If A *⇒ φ *⇒ α for some A ∈ N, we say φ derives α, and φ, α are called derived sentences. We will display derivations as trees. These "derivation trees" impose a structure on the words of a language that is useful in applications such as compilation and translation of programming languages.

The nodes of a derivation tree are labeled with terminal and nonterminal symbols of the grammar. If an interior node m is labeled A, and the sons (subtrees) of m are labeled b1, ..., bj from the left, then A → b1 ... bj must be a production. Furthermore, if t is a derivation tree, root (t) = Z, and frontier (t) = w then A *⇒ w. A derivation tree t is associated with a "leftmost" derivation, and
this particular derivation is sometimes denoted \( Z \xrightarrow{t} w \), the leftmost derivation of \( w \) from \( Z \) via \( t \). A more detailed treatment of this method may be found in [20].

2.3. Algebraic Syntax and Semantics

The following definitions are needed for developing algebraic syntax and semantics. These are similar to the definitions presented by Goguen, Thatcher, and Wagner [11].

Let \( N \) be a nonempty set, called the set of "sorts." The letter \( N \) will be used throughout this paper for sorts, \( N^* \) denotes the set of all finite words over \( N \), and the empty word being \( \lambda \). Concatenation of \( n \) letters \( s_1, \ldots, s_n \in N \) is denoted by \( s_1 \ldots s_n \in N^* \). An \( N \)-sorted set \( A \) is an indexed family \( \{A_n/n \in N\} \). If \( w = s_1 \ldots s_m \in N^* \), then \( A^w \) denotes that set \( A_{s_1} \times \ldots \times A_{s_m} \). In particular \( A^n \) for \( n \in N \) denotes \( A_n \), and \( A^\lambda \) is the singleton \( \{\emptyset\} \). \( A_n \) is called the "carrier" (underlying set) of sort \( n \). The carrier \( A_n \) can be thought of as having the name "\( n \)."

The following definition of \( N \)-sorted signature is key in our development.

**Definition 2.1** [\( N \)-sorted Signature]

Let \( N \) be a set of sorts. Then, an alphabet \( \Sigma \) is an \( N \)-sorted signature if \( \Sigma = \{\Sigma(w,n)/w \in N^*, n \in N\} \) where the \( \Sigma(w,n) \)'s are disjoint subsets of \( \Sigma \) such that only finitely many of them are nonempty. If \( f \in \Sigma(w,n) \), then we say that an operator (function) \( f \) has rank \( |w| \) (the length of \( w \)) and sort \( n \in N \).
Example 2.2

Let \( N = \{\text{Set, Integer, Boolean}\} \) and \( \Sigma \) be an \( N \)-sorted signature as follows (abbreviating Set, Integer, and Boolean as \( S, I, \) and \( B \)):

\[
\begin{align*}
\Sigma(\lambda, B) &= \{\text{TRUE, FALSE}\}, \quad \Sigma(\lambda, I) = \{\text{ZERO}\}, \quad \Sigma(I, I) = \{\text{SUCC, PRED}\} \\
\Sigma(\lambda, S) &= \{\text{NULL}\}, \quad \Sigma(IS, S) = \{\text{ADD, DELETE}\}, \quad \Sigma(S, B) = \{\text{EMPTY}\} \\
\Sigma(IS, B) &= \{\text{MEMBER}\}, \quad \Sigma(SS, S) = \{\text{UNION}\}, \quad \Sigma(S, I) = \{\text{SIZE}\} \\
\Sigma(SS, B) &= \{\text{EQUAL}\}, \quad \Sigma(BSS, S) = \{\text{IF}\}.
\end{align*}
\]

We will later see the "meaning" of \( \Sigma \) precisely, but for the moment it suffices to say that the members of \( \Sigma \) are operation names for a "set" data structure.

The basic syntactic information about a data structure is conveyed by its signature, which gives the sorts, the operation names, and the sorts of their arguments and values. Syntactic information of a data structure can be expressed in a graphical form, in which each sort is a node, and each operation is a "polyedge", drawn as an arrow whose tails come from the arguments of \( \Sigma \), and whose head go to the results sorts (codomain) of the operation. For example, the signature of "set" data structure (Example 2.1) appears in Figure 2.1. We now need to define an \( N \)-sorted \( \Sigma \)-algebra. Basic definitions and results concerning \( N \)-sorted (many-sorted) algebras can be found in [4].

Definition 2.3 \( [N\text{-sorted } \Sigma \text{-algebra}] \)

Let \( N \) be a set of sorts. Then an "\( N \)-sorted \( \Sigma \)-algebra" is a pair \( <A, F> \) such that \( A \) is an \( N \)-indexed family of sets \( \{A_n/n \in N\} \)
Figure 2.1 Signature for "set" data structure
(each $A_n$ is called the underlying set), and $F$ is a $\Sigma$-indexed family of functions \{f/y^\_f \in \Sigma(w,n)\} such that if $f \in \Sigma(w,n)$ then $f^\_A : A^w \rightarrow A_n$ (where $A^w = A_{s_1} \times \ldots \times A_{s_m}$ for $w = s_1 \ldots s_m$). However, if $g \in \Sigma(\lambda,n)$ then $g^\_A : \{\emptyset\} \rightarrow A_n$ and we write $g^A$ for $g^A(\emptyset) \in A_n$. We will often refer to a $\Sigma$-algebra $<A,F>$ by $A$ if the operation set $F$ is understood.

**Example 2.4**

Let $A$ be the following algebra, and $\Sigma$ be defined as Example 2.1. The underlying set for algebra $A$ are $A_S$, $A_I$, $A_B$ where

- $A_B = \{\text{true, false}\}$,
- $A_I = \{0, 1, -1, 2, -2, \ldots\}$, and
- $A_S = P(A_I)$, the powerset of $A_I$ by defining its functions (operations), we have:

$$F = \{\text{NULL}_A, \text{ADD}_A, \text{DELETE}_A, \text{UNION}_A, \text{EMPTY}_A, \text{MEMBER}_A, \text{EQUAL}_A, \text{SIZE}_A, \text{IF}_A, \text{ZERO}_A, \text{SUCC}_A, \text{PRED}_A\}$$

where

- $\text{NULL}_A : \{\emptyset\} \rightarrow A_S$
- $\text{NULL}_A(\emptyset) = \text{NULL}_A = \emptyset$
- $\text{ADD}_A : A_I \times A_S \rightarrow A_S$
- $\text{ADD}_A(x, \{a_1, \ldots, a_m\}) = \{a_1, \ldots, a_m\} \cup \{x\}$
- $\text{DELETE}_A : A_I \times A_S \rightarrow A_S$
- $\text{DELETE}_A(x, \{a_1, \ldots, a_m\}) = \{a_1, \ldots, a_m\} - \{x\}$
UNION_A: A_S × A_S → A_S
UNION_A \((\{x_1, \ldots, x_m\}, \{y_1, \ldots, y_k\}) = \{x_1, \ldots, x_m\} \cup \{y_1, \ldots, y_k\}\)

EMPTY_A: A_S → A_B
EMPTY_A(\{a_1, \ldots, a_m\} = \{false if m > 0\}

MEMBER: A_I × A_S → A_B
MEMBER_A(x, \{a_1, \ldots, a_m\} = \{true if x ∈ \{a_1, \ldots, a_m\}\}

EQUAL_A: A_S × A_S → A_B
EQUAL_A(\{a_1, \ldots, a_m\}, \{b_1, \ldots, b_k\} = \{true if \{a_1, \ldots, a_m\} = \{b_1, \ldots, b_k\}\}

SIZE_A: A_S → A_I
SIZE_A(\{a_1, \ldots, a_m\} = \{m if m > 0\}

IF_A: A_S × A_S × A_S → A_S
IF_A(b, \{a_1, \ldots, a_m\}, \{c_1, \ldots, c_k\} = \{\{a_1, \ldots, a_m\}\} if b = true

ZERO_A: \{\emptyset\} → A_I
ZERO_A(\emptyset) = ZERO_A = 0

SUCC_A = A_I → A_I
SUCC_A(x) = x + 1

PRED_A = A_I → A_I
PRED_A(x) = x - 1

\(\varepsilon\) of Example 2.2 can be represented by the following cfg G (Figure 2.2). We will demonstrate formally that there exists a signature (\(\varepsilon\)) for every cfg. The cfg G of Figure 2.2 along with the "semantic
rules" as in Example 2.4 are said to form a "language definition system" (LDS). At the end of this chapter, we will provide the formal definition of a language definition system.

In Chapter 1, we mentioned that algebraic tree transducers are crucial to our developments. Algebraic tree transducers will be determined by means of certain mappings between algebras, called "homomorphisms", defined below.

Definition 2.5 [N-sorted $\Sigma$-homomorphism]

Let $A$ and $B$ be two $N$-sorted $\Sigma$-algebras, and let $h = \{h_s : A_s \rightarrow B_s / s \in N\}$ be an $N$-indexed family of functions (i.e., $h : A \rightarrow B$). Then $h : A \rightarrow B$ is a "$\Sigma$-homomorphism" if the following conditions are satisfied:

- (H0) if $f \in \Sigma(\lambda, s)$ then $h_s (f) = f$
- (H1) if $f \in \Sigma(w, s)$, $w = s_1 \ldots s_m$, $a_i \in A_{s_i}$,

then $h_s (f(a_1, \ldots, a_m)) = f(h_{s_1}(a_1), \ldots, h_{s_m}(a_m))$. □

Representing $h^w = h_{s_1} \times \ldots \times h_{s_m}$ for $w = s_1 \ldots s_m$, then the definition of $\Sigma$-homomorphism may be pictorially represented as the commuting diagram in Figure 2.3.

We introduce an "initial algebra" concept that enables us to characterize an object "abstractly," or independent of representation, only in terms of its structure. In our development, we will consider syntax of a language as an initial algebra in a class of algebras. Initiality of an algebra will also enable us to find a
cfg G = (N, Δ, P, Z) where

\[ N = \{S, I, \emptyset\}, \quad Z = \{\emptyset\}, \]

\[ \Delta = \{\text{add}, \text{delete}, \text{union}, \text{if}, (,), \text{then}, \text{else}, \text{empty}, \text{member}, \text{equal}, \text{true}, \text{false}, 0, 1, +, -, ', '\} \]

and P is given below:

<table>
<thead>
<tr>
<th>Name of production</th>
<th>Actual production</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>S→\lambda</td>
</tr>
<tr>
<td>ADD</td>
<td>S→add(I, S)</td>
</tr>
<tr>
<td>DELETE</td>
<td>S→delete(I, S)</td>
</tr>
<tr>
<td>UNION</td>
<td>S→union(S, S)</td>
</tr>
<tr>
<td>IF</td>
<td>S→if(B) then S else S</td>
</tr>
<tr>
<td>EMPTY</td>
<td>B→empty(S)</td>
</tr>
<tr>
<td>MEMBER</td>
<td>B→member(I, S)</td>
</tr>
<tr>
<td>EQUAL</td>
<td>B→equal(S, S)</td>
</tr>
<tr>
<td>TRUE</td>
<td>B→true</td>
</tr>
<tr>
<td>FALSE</td>
<td>B→false</td>
</tr>
<tr>
<td>ZERO</td>
<td>I→0</td>
</tr>
<tr>
<td>SUCC</td>
<td>I→I+1</td>
</tr>
<tr>
<td>PRED</td>
<td>I→I-1</td>
</tr>
</tbody>
</table>

Figure 2.2 The cfg for "set" data structure
unique meaning of sentences of a context-free grammar. Now, we give the definition of an initial algebra, then, we give the definition of an important algebra that is initial. Also, the class of \( \Sigma \)-algebras will be denoted by \( \text{ALG}(\Sigma) \).

**Definition 2.6** [Initial Algebra]

\( \Sigma \)-algebra \( A \) is "initial", if for every (any other) \( \Sigma \)-algebra \( B \) there is a unique homomorphism \( h: A \rightarrow B \).

Thus, initiality can be used to capture a particular structure abstractly, that is, independently of implementation.

**Definition 2.7** \([T_{\Sigma}]\)

\( T_{\Sigma} \) is a \( \Sigma \)-algebra defined as below. We also concurrently define the lengths of elements of \( T_{\Sigma} \).

1. \( f_{T_{\Sigma}} \in f \in T_{\Sigma} \) if \( f \in \Sigma(\lambda, s) \) and \( |f| = 1 \).

2. \( f_{T_{\Sigma}}(t_1, ..., t_n) = f_{<t_1 ... t_n>} \in T_{\Sigma} \) if \( f \in \Sigma(w, s) \), \( w = s_1 ... s_n \), \( t_i \in T_{\Sigma} \) for \( i = 1, ..., n \), and \( |f_{<t_1 ... t_n>}| = 1 + |t_1| + ... + |t_n| \).

3. nothing else in \( T_{\Sigma} \) other than elements obtained as above.

The elements of \( T_{\Sigma} = \{T_{\Sigma} / s \in \mathbb{N}\} \) are called trees. Thus, a tree consists of a labeled node, such that a node labeled with \( f \in \Sigma(w, s) \) has \( |w| \) sons (subtrees). If \( f \in \Sigma(w, s) \) is a root of tree \( t \) then \( t \) is of \( s \).
Figure 2.3 Commutative diagram: homomorphism
Example 2.8
Let $N = \{I\}$ and $\Sigma$ be an $N$-sorted signature where
$\Sigma(I, I) = \{a, b\}$, $\Sigma(\lambda, I) = \{f, g\}$, $\Sigma(I, \lambda) = \{h\}$
Then $h < f < a < h < g < b > e T \Sigma$ and is represented pictorially in Figure 2.4.
Theorem 2.9 shows that $T \Sigma$ is initial in $\text{ALG}(\Sigma)$. □

Theorem 2.9
$T \Sigma$ is an initial algebra.

Proof: It suffices to prove that there is a unique homomorphism $h: T \Sigma \rightarrow A$ for any given $\Sigma$-algebra $A$. We show that $h_s(t)$ is uniquely defined for all $s \in N$ and for $t \in T \Sigma_s$ on the length of $t$. If $|t| = 1$,
then $t = f \in \Sigma(\lambda, s)$, hence, $h_s(t) = h_s(f) = f_A$.

Now suppose that the hypothesis is true for all $s \in N$, $t \in T \Sigma_s$ with $|t| < m$. Now consider $t \in T \Sigma_s$ with $|t| = m$. Hence, $h_s(t) = h_s(f_{<t_1 \ldots t_n>})$
for some integer $n$, and some $f \in \Sigma(w, s)$ where $w = s_1 \ldots s_n$ and $t_i \in T \Sigma_{s_i}$.

Hence, $h_s(t) = h_A(h_{t_1}(s_1), \ldots, h_{t_n}(s_n))$. By induction hypothesis
each $h_{t_i}(s_i)$ is uniquely defined, hence $h_s(t)$ is uniquely defined. □

Since one of our aims is to translate between two context-free languages, we define signature of context-free grammars. If $\Sigma$ is a signature of cfg $G = (N, \Delta, P, Z)$, then each element of $\Sigma$ corresponds to an element of $P$ (i.e., is the "name" of a production).
Figure 2.4. Pictorial representation of $h<f\langle a\rangle h<g\langle b\rangle b\rangle$
Definition 2.10 [Signature of a Context-Free Grammar]

Let \( G = (N, \Delta, P, Z) \) be a context-free grammar. Then the signature of \( G \) is \( \Sigma_G = \{ \Sigma_G(w,s)/w \in N^*, s \in N \} \), where for all \( p \in P \), if \( p = B \rightarrow b_0 B_1 b_1 \ldots B_k b_k \) with \( B, B_1, \ldots, B_k \in N \) and \( b_0, b_1, \ldots, b_k \in A \), then \( p \in \Sigma_G(w,s) \) where \( w = B_1 \ldots B_k \) and \( s = B \).

The elements of \( \Sigma_G \) (i.e., \( \Sigma_G(w,s) \)) are disjoint for a cfg \( G \), because each production of \( G \) has a unique name. For a cfg \( G \), \( T_{\Sigma_G} \) will be called the set of "concrete syntax trees".

An important algebra that is concerned with cfg's is the "string" \( \Sigma \)-algebra. Let \( \Sigma_G \) be the signature of cfg \( G = (N, \Delta, P, Z) \). Then the string \( \Sigma_G \)-algebra \( ST \) is a pair \( \{ST_n/n \in N\}, \{P_{ST}/p \in P\} \) where \( ST_n \in \Delta^* \) for all \( n \in N \). If \( p \in P \) is \( B \rightarrow b_0 B_1 b_1 \ldots B_k b_k \) where \( b_0, b_1, \ldots, b_k \in A \), and \( B, B_1, \ldots, B_k \in N \) and if \( x = (x_1, \ldots, x_k) \in ST_{B_1} x \ldots x_{ST_{B_k}} \), then \( p_{ST}(x) = b_0 x_1 b_1 \ldots x_k b_k \).

Now consider the unique homomorphism \( \delta: \Sigma_T \rightarrow ST \) such that \( \Sigma \) is a signature for some cfg \( G \) and \( ST \) is a string \( \Sigma \)-algebra. Then for all \( n \in N \) and \( t \in T_\Sigma \), \( \delta_n(t) = w \) iff \( n \xrightarrow{T} w \), where \( n \xrightarrow{T} w \) denotes that \( w \) is the result of a leftmost derivation from \( n \) via \( t \). We will call \( \delta \) a "concrete syntax homomorphism" and \( \delta(T_\Sigma) = \{\delta_n(T_\Sigma)/n \in N\} \) is the set of sets of strings derived from various nonterminals of cfg \( G \). The next example illustrates the concept of string algebra.
Example 2.11

Consider cfg G of Figure 2.2. The string $\varepsilon$-algebra of cfg G is

$$ST = \langle ST_s, ST_I, ST_B \rangle, (p_{ST}/p \in P) \text{ with } ST_I = ST_B = ST_S = \Delta^*.$$ The function $p_{ST}$ for $p \in P$ are given below for some productions, and the symbol $||$ is a concatenation operator.

- **NULL$_{ST}$**: $\{\emptyset\} \rightarrow ST_s$ with $\text{NULL}_{ST}(\emptyset) = \lambda$
- **ADD$_{ST}$**: $ST_I \times ST_S \rightarrow ST_S$ with $\text{ADD}_{ST}(x, y) = \text{'add ('}||x||', '||y||')'$
- **IF$_{ST}$**: $ST_B \times ST_S \times ST_S \rightarrow ST_S$ with $\text{IF}_{ST}(b, x, y) = \text{'if '||b||'then '||x||' else '||y}$$
- **ZERO$_{ST}$**: $\{\emptyset\} \rightarrow ST_I$ with $\text{ZERO}_{ST}(\emptyset) = '0'$
- **SUCC$_{ST}$**: $ST_I \rightarrow ST_I$ with $\text{SUCC}_{ST}(x) = x' ||' + 1$

The construction of $p_{ST}$ for the remaining productions is left to the reader.

The unique homomorphism $\delta: T_\varepsilon \rightarrow ST$ has:

- $\delta_s(\text{NULL}_{T_\varepsilon}) = \text{NULL}_{ST}$
- $\delta_s(\text{ADD}_{T_\varepsilon}(x_1, x_2)) = \text{ADD}_{ST}(\delta_I(x_1), \delta_I(x_2))$
- $\delta_I(\text{SUCC}_{T_\varepsilon}(x)) = \text{SUCC}_{ST}(\delta_I(x))$

The construction of the remaining homomorphisms is left to the reader.

For $t = \text{ADD} < \text{ZERO} \text{ ADD} < \text{SUCC} < \text{ZERO} > \text{NULL} >> \epsilon_{T_\varepsilon}$ we have:

$\delta_s(t) = \delta_s(\text{ADD} < \text{ZERO} \text{ ADD} < \text{SUCC} < \text{ZERO} > \text{NULL} >>)$

- $= \text{ADD}_{ST}(\text{ZERO}_{ST}, \text{ADD}_{ST}(\text{SUCC}_{ST}(\text{ZERO}_{ST}), \text{NULL}_{ST}))$
- $= \text{ADD}_{ST}(0, \text{ADD}_{ST}(\text{SUCC}_{ST}(0), \lambda))$
- $= \text{ADD}_{ST}(0, \text{ADD}_{ST}(0+1, \lambda))$
- $= \text{ADD}_{ST}(0, \text{ADD}_{ST}(0+1, \lambda))$
- $= "\text{add}(0, \text{add}(0+1, \lambda))"$
We should note that "add (0, add (0+1, X))" is the frontier of the derivation tree t of cfg G. Furthermore, the set of sentences generated by G is \( \{ \delta(x) | x \in T_{\Sigma} \} \).

Previously, we mentioned that a "language" is a set of pairs, sentences along with their meanings. So far, we have discussed syntactic issues about languages. Now, we define another important homomorphism, the "semantic homomorphism" that deals with semantics of context-free grammars. The notion behind semantics is to give meanings to every sentence of context-free grammar G. For finding the "meaning" of a sentence, \( T_{\Sigma} \) (where \( \Sigma \) is the signature of G) plays an important role, because \( T_{\Sigma} \) is an initial algebra. Hence, for each sentence of cfg G there is a unique way of finding the meaning using the unique homomorphism \( \mu: T_{\Sigma} \rightarrow A \), where A is an algebra for semantics. Now we have enough machinery to give the formal definition of a "language definition system" (LDS), which is a pair of syntax and semantics.

**Definition 2.12** [Language Definition System]

Let G be a cfg with associated signature of \( \Sigma \). Let A be an arbitrary \( \Sigma \)-algebra. Then a "Language definition system" (LDS) is a pair \( D = (G, A) \), whose algebraic semantics is \( A \). The algebra \( T_{\Sigma} \) (where \( \Sigma \) is a signature of cfg G) along with the semantic rules (the definition of \( \mu: T_{\Sigma} \rightarrow A \) is called the set of rules for obtaining the meaning of each syntactic structure) is said to form an LDS, and the Language of an LDS is a set of pairs \( (\delta(t), \)
2.4. Derived and Represented Algebras

Here we need to define two important signatures and their associated algebras, which will be used in construction of algebraic tree transducers in Chapter 4. Derived signatures enable us to have complex trees (trees of height greater than one) in construction of tree transducers. Represented signatures enable us to translate languages whose underlying grammars are of different signatures.

Definition 2.13 [Derived Signature]

Let $\Sigma = \{\Sigma(w, n)/w \in \mathbb{N}^*, n \in \mathbb{N}\}$ be an $\mathbb{N}$-sorted signature. Then the "derived signature" of $\Sigma$ is $\tilde{\Sigma} = \{\tilde{\Sigma}(w, n)/w \in \mathbb{N}^*, n \in \mathbb{N} \text{ with } \mathbb{N} = \mathbb{N} \cup \{\lambda\}\}$, which is the smallest set such that

1. $\{X_1^\alpha/0 \leq i \leq k, \alpha = \alpha_1 \ldots \alpha_k \in \mathbb{N}^*\} \subseteq \tilde{\Sigma}$

   with $\begin{cases} X_i^\alpha \in \tilde{\Sigma}(\alpha, \alpha_i) & \text{if } i > 0 \\ X_0^\alpha \in \tilde{\Sigma}(\alpha, \lambda) & \text{if } i = 0 \end{cases}$

2. if $q \in \Sigma(\lambda, s)$ then $q[X_1^\alpha, ..., X_k^\alpha] \in \bar{\Sigma}(\alpha, s)$; and if $q \in \Sigma(s_1 \ldots s_k, s)$ then $q[X_1^\alpha, ..., X_k^\alpha] \in \bar{\Sigma}(\alpha, s)$ where $\alpha_1 \ldots \alpha_k = s_1 \ldots s_k$
Figure 2.5 Syntax and semantics homomorphisms
(3) if $q_0 \in \Sigma(w, s)$ and $q_i \in \Sigma(w, \alpha_i)$ for $i = 1, \ldots, k$ where $\alpha = \alpha_1 \ldots \alpha_k$, then $q = q_0[q_1 \ldots q_k] \in \Sigma(w, s)$.

An element $X_i^\alpha$ of $\Sigma$ is a "variable" of sort $\alpha_i$. Let $\Sigma$ be the derived signature of $\Sigma$, since $\Sigma$ is a signature, it gives rise to be a class of $\Sigma$-algebras. $\Sigma$-algebras derived from $\Sigma$-algebras play an important role in computing the meaning of "derivation trees." We now give the formal definition of derived algebra.

**Definition 2.14 [Derived Algebras]**

Let $A$ be an $N$-sorted $\Sigma$-algebra. Then the derived $\bar{N}$-sorted $\Sigma$-algebra $\bar{A}$ is $\langle \bar{A}_n/\bar{n} \in \bar{N} \rangle, \{f^\bar{A}/fe\Sigma\}$

where

(a) $\bar{A}_n = A_n$ for $n \in N$ and $\bar{A}_\lambda = \emptyset$

(b) if $f \in \Sigma(\alpha, s)$ then $f^\bar{A}: \bar{A}_{\alpha} \rightarrow \bar{A}_s$, moreover if $y = (y_1, \ldots, y_k)$ is an element of $\bar{A}_n$, then

1. $x^\alpha_{0, \bar{A}}(y) = \emptyset$
2. $x^\alpha_{1, \bar{A}}(y) = y_i$ (a variable of sort $\alpha_i$); and $q \bar{A}(x^\alpha_{i_1} \ldots x^\alpha_{i_k}) = q_{\bar{A}}(y_{i_1}, \ldots, y_{i_k})$.

From now on we will write $A_n$ instead of $\bar{A}_n$.

Note that any (derived) operation of $\Sigma$ has a uniquely determined "normal" form. To expand, if $\bar{A}$ is a derived algebra and if $P, P', P''$ are elements of $\bar{A}$ such that $Ze\bar{N}, ZZZZ = \alpha, ZZZ = \alpha'$; and
P = a[x_1 x_2 x_3]^\alpha \ [b[x_1 x_2]^\alpha \ c[x_3]^\alpha \ x_4^\alpha] \\
P' = a[x_1 x_2 x_3]^\alpha' \ [b[x_1 x_2]^\alpha \ c[x_3]^\alpha \ x_4^\alpha] \\
P'' = a[b[x_1 x_2]^\alpha \ x_3^\alpha x_4^\alpha] \ [x_1^\alpha x_2 \ c[x_3]^\alpha \ x_4^\alpha] \\

Then \( P_A (y_1, y_2, y_3, y_4) = P_A' (y_1, y_2, y_3, y_4) \)
\hspace{3cm} = P_A'' (y_1, y_2, y_3, y_4) \\

So, we use \( P \) as the "normal" form of the derived operations of \( P, P', \) and \( P''. \) The normal form derivation does not need to have the \( x_1^\alpha \) written out. So,
\( a[x_1 x_2 x_3]^\alpha \ [b[x_1 x_2]^\alpha \ c[x_3]^\alpha \ x_4^\alpha] = a[b[x_1 x_2]c[x_3] x_4^\alpha] \)

Thus, we use the following correct form of elements of \( \Xi \) as:

1. \( \{x_i^\alpha/0 \leq i \leq k, \alpha = \alpha_1 ... \alpha_k \in \mathbb{N}^* \} \subseteq \Xi \)
   
   with \( \{x_i^\alpha \in \Xi (\alpha, \alpha_i) \text{ if } \alpha > 0 \}
   \begin{cases} 
   x_0^\alpha \in \Xi (\alpha, \lambda) \text{ if } \alpha = 0 \n   \end{cases} \)

2. if \( q \in \Xi (\lambda, s) \) then \( q[x_0^\alpha] \in \Xi (\lambda, s); \) and if \( q \in \Xi (s_1...s_k, s) \) then
   \( q[x_1^\alpha ... x_k^\alpha] \in \Xi (\alpha_i, s) \) where \( \alpha_1 ... \alpha_k = s_1 ... s_k. \)

3. if \( q_0 \in \Xi (w, s) \) and \( q_i \in \Xi (w, \alpha_i) \) for \( i = 1, ... , k \) where \( \alpha = \alpha_1 ... \alpha_k \) then \( q = q_0[q_1 ... q_k] \in \Xi (w, s). \)

However, note that the formal definition of \( \Xi \) gives the second type of definition.
Example 2.15

To illustrate the use of derived algebras, consider the following algebra $A$. Given $N = \{\text{Integer, Boolean}\} = \{I, B\}$, let $\Sigma$ be defined by:

$$\Sigma(\lambda, I) = \{\text{ZERO}\}, \Sigma(\lambda, B) = \{\text{TRUE}, \text{FALSE}\}$$

$$\Sigma(I, I) = \{\text{ADD}, \text{SUB}\}, \Sigma(I, B) = \{\text{EQ}\}, \Sigma(B, I) = \{\text{IF}\}.$$ 

Thus, the underlying set for $A$ is $\{A_I, A_B\}$ with $A_B = \{\text{true, false}\}$, $A_I = \{0, 1, -1, 2, -2, \ldots\}$, and $F = \{\text{ZERO}, \text{ADD}, \text{SUB}, \text{EQ}, \text{IF}, \text{TRUE}, \text{FALSE}\}$

where

- $\text{ZERO}_A: \emptyset \to A_I$ with $\text{ZERO}_A(\emptyset) = \text{ZERO} = 0$
- $\text{ADD}_A: A_I \times A_I \to A_I$ with $\text{ADD}_A(x_1, x_2) = x_1 + x_2$
- $\text{SUB}_A: A_I \times A_I \to A_I$ with $\text{SUB}_A(x_1, x_2) = x_1 - x_2$
- $\text{EQ}_A: A_I \times A_I \to A_B$ with $\text{EQ}_A(x_1, x_2) = \begin{cases} \text{true} & \text{if } x_1 = x_2 \\ \text{false} & \text{if } x_1 \neq x_2 \end{cases}$
- $\text{IF}_A: A_B \times A_I \times A_I \to A_I$ with $\text{IF}_A(b, x_1, x_2) = \begin{cases} x_1 & \text{if } b = \text{true} \\ x_2 & \text{if } b = \text{false} \end{cases}$
- $\text{TRUE}_A: \emptyset \to A_B$ with $\text{TRUE}_A(\emptyset) = \text{TRUE} = \text{true}$
- $\text{FALSE}_A: \emptyset \to A_B$ with $\text{FALSE}_A(\emptyset) = \text{FALSE} = \text{false}$

The derived signature of $\Sigma$ is $\Sigma = \{X_{II}, X_{II}, X_{II}, \ldots,\}$, $\text{ADD}(X_1, X_2), \text{IF}(X_1, \text{ZERO}(X_0), X_3, \ldots)$. Thus, if $R \in \Sigma$, then $R$ is an ordered tree with elements of $\Sigma$ labeling the branching nodes and various $X_i$ labeling the leaf nodes. Since $\alpha$ is the same for all leaf nodes, it need only once in a derived operation. For the sake
of simplicity, we can eliminate all occurrences of \(X_0\) which appear directly under nullary element of \(\Sigma\). Thus

\[
\text{ZERO}_{\Sigma}^\text{BII} = \text{ZERO}_{\Sigma} = \text{ZERO}, \quad \text{ADD}_{\Sigma}^\text{BII} = \text{ADD}_{\Sigma}^\text{II}, \quad \text{and} \quad \text{IF}_{\Sigma}^\text{BII} = \text{IF}_{\Sigma} = \text{IF}_{\Sigma}^\text{II}.
\]

The \(\Sigma\)-algebra \(\tilde{A}\) derived from \(\Sigma\)-algebra \(A\) is

\[
\tilde{A} = \langle \{A_1, A_2\}, \{f_A, f_{\Sigma}\} \rangle
\]

where \(f_A\) is an operation as determined by Definition 2.14. In order to exemplify the concept of derived algebra \(\tilde{A}\), we compute the following operations:

\[
\text{ADD}_{\tilde{A}}^\text{II}(x_1, x_2) = \text{ADD}_{\tilde{A}}^\text{II}(x_1, x_2) = \text{ADD}(x_1, x_2) = x_1 + x_2, \quad \text{and}
\]

\[
\text{IF}_{\tilde{A}}^\text{BII}(x_3, x_2) = \text{IF}(\text{true}, \text{SUB}(x_3, x_2), x_2) = x_3 - x_2.
\]

The following list of identities is true in every derived \(\Sigma\)-algebra \(\tilde{A}\).
1) \( X_0^{\alpha}(p_1 ... p_k)_{\bar{A}} = X_0^{\alpha} \) whenever \( p_i \in \Sigma(\beta, \alpha_i) \) and \( \alpha = \alpha_1 ... \alpha_k \)

2) \( X_1^{\alpha}(p_1 ... p_k)_{\bar{A}} = p_1,_{\bar{A}} \)

3) \( P[p_1 ... p_k](q_1 ... q_m)_{\bar{A}} = P[p_1(q_1 ... q_m) ... p_k(q_1 ... q_m)]_{\bar{A}} \)

4) \( P[X_1^{\alpha} ... X_k^{\alpha}]_{\bar{A}} = P_{\bar{A}} \) whenever \( P \in \Sigma(\alpha, n) \) for some \( n \in \mathbb{N} \).

Since we will be working with trees, the following definitions are necessary to distinguish the terminated trees from the nonterminated trees, which will be used in the next chapters.

**Definition 2.16** [Terminated Tree, Nonterminated Tree]

Let \( \Sigma \) be a signature and \( \bar{\Sigma} \) be the derived signature of \( \Sigma \).

If \( t \in T_{\Sigma} \) for some sort \( s \) then \( t \) is called a "terminated tree" (a tree which has no variables in it). The symbols \( t, t', t'', \ldots, t_0, t_0', t_0'', \ldots, t_1, t_1', t_1'', \ldots \) will be used for terminated trees. The symbols \( T, T_0, T_1, T_2, \ldots \) will be used to denote sets of terminated trees. If \( f \in \bar{\Sigma}(w, s) \), then \( f \) is called a "nonterminated tree" (a tree which has variables in it). The symbols \( f, f', f'', \ldots, f_0, f_0', f_0'', \ldots, f_1, f_1', f_1'', \ldots, P, P', P'', \ldots, P_0, P_0', P_0'', \ldots, q_0, q_0', q_0'', \ldots, q_1, q_1', q_1'', \ldots \) will be used for nonterminated trees. The symbols \( \Theta_1, \Theta_2, \ldots \) will be used to denote a set of nonterminated trees.

Thus, if \( f \in \bar{\Sigma}(w, s) \), \( w = s_1 ... s_n \), and \( t_i \in T_{\Sigma} \) for \( i = 1, \ldots, n \) then \( f_{\bar{T}_\Sigma}(t_1, \ldots, t_n) \) is a terminated tree. In Theorem 2.9 we proved that there is a unique homomorphism from \( T_\Sigma \) to \( A \). In a similar way,
it has been shown in [31] that there is a unique homomorphism from $T_\Sigma$ to $\bar{A}$, if $\bar{A}$ and $\bar{\Sigma}$ are derived algebra of $A$ and derived signature of $\Sigma$ respectively. Also, in [31] they have proved that if $A$ and $B$ are $\Sigma$-algebras, $\bar{A}$ and $\bar{B}$ are derived algebras of $A$ and $B$ respectively, then $h$: $A\rightarrow B$ is a homomorphism if and only if $h$: $\bar{A}\rightarrow \bar{B}$ is a homomorphism where $h = h_\lambda: \{\emptyset\} \rightarrow \{\emptyset\}$. In order to be complete, we present these results below.

**Theorem 2.17**

Let $\Sigma$ be an $N$-sorted signature with derived signature $\bar{\Sigma}$. Let $A$ and $B$ be $\Sigma$-algebras, and $\bar{A}$ and $\bar{B}$ be their derived $\bar{\Sigma}$-algebras. Then $h$: $A\rightarrow B$ is a homomorphism if and only if $h = h_\lambda: \{\emptyset\} \rightarrow \{\emptyset\}$ is a homomorphism from $\bar{A}$ to $\bar{B}$.

**Proof:** Only if: The proof that $h$: $\bar{A}\rightarrow \bar{B}$ is a homomorphism given that $h$: $A\rightarrow B$ is a homomorphism is by induction on the height of $f\in \Sigma$.

We define the height of $f(h(f))$ to be 0 if $f = x_i^g$ and $1 + \max(h(f_i), 1 \leq i \leq k)$ if $f = f_0[f_1 \ldots f_k]$. It can be shown by induction on $h(f)$ that $h_\lambda(f^A(x)) = f^B(h^A(x))$ where $f\in \Sigma(a,n)$.

If: The proof that $h$: $A\rightarrow B$ is a homomorphism given that $h$: $\bar{A}\rightarrow \bar{B}$ is a homomorphism is obvious from the statement of the theorem. □

In order to construct tree transducer algebraically, we need to define "represented signature" and the associated "represented algebra". These two concepts enable us to translate from one algebra
to another, which have different signatures, overcoming the limitations of homomorphisms which are only defined between algebras of the same signature. Represented signatures will be used for two purposes: first, they enable us to translate between languages whose underlying grammars are of different signatures; second, they enable us to implement a new data structure in terms of other data structures.

**Definition 2.18 (Represented Signature)**

Let $\Sigma$ and $\Sigma'$ be $N$-sorted and $N'$-sorted signatures, respectively. Let $n: N' \rightarrow N$ with $n(\lambda) = \lambda$, $n(s_1 ... s_n) = n(s_1) ... n(s_n)$, and $\Pi: \Sigma' \rightarrow \Sigma$ be given functions. If for each $f \in \Sigma'(w,s)$ we have $\Pi(f) \in \Sigma(n(w), n(s))$, then $\Sigma'$ is said to be represented in host signature $\Sigma$. □

Now we give the definition for a "represented algebra", which will be used in the construction of tree transducers discussed in Chapter 4.

**Definition 2.19 (Represented Algebra)**

Let $\Sigma'$ be represented in host signature of $\Sigma$ via representation functions of $\eta$ and $\Pi$. Let $A$ be an $N$-sorted $\Sigma$-algebra. Then the $N'$-sorted $\Sigma'$-algebra $<A',F'>$ is said to be represented in $A$ where $A' = \{A_n / n = n(n') \text{ and } n' \in N'\}$, and $F' = \{f_A / f = \Pi(f') \text{ and } f' \in \Sigma'(w',n')\}$. □

**Example 2.20**

This example illustrates the concept of represented signature. Let $\Sigma_1, \Sigma_2$, and $\Sigma'$ be signatures such that $\Sigma'$ is represented in $\Sigma_j$.
(i=1,2) with \( \eta_1(I)=E, \eta_2(I)=F, \) and \( \eta_1(C)=\eta_2(C)=B. \)

<table>
<thead>
<tr>
<th>( \Sigma_1 ) signature</th>
<th>( \Sigma_2 ) signature</th>
<th>( \Sigma' ) signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma_1(EE,E) = {ADD,MUL} )</td>
<td>( \Sigma_2(FF,F) = {SUB,DIV} )</td>
<td>( \Sigma'(II,I) = {a,b} )</td>
</tr>
<tr>
<td>( \Sigma_1(EE,B) = {EQ} )</td>
<td>( \Sigma_2(FF,B) = {LE} )</td>
<td>( \Sigma'(II,C) = {c} )</td>
</tr>
<tr>
<td>( \Sigma_1(BEE,E) = {IF} )</td>
<td>( \Sigma_2(\lambda,F) = {ZERO,ONE} )</td>
<td>( \Sigma'(\lambda,I) = {d} )</td>
</tr>
<tr>
<td>( \Sigma_1(\lambda,E) = {ONE} )</td>
<td>( \Sigma_2(BB) = {AND} )</td>
<td>( \Sigma'(\lambda,C) = {e,f} )</td>
</tr>
<tr>
<td>( \Sigma_1(\lambda,B) = {TRUE,FALSE} )</td>
<td>( \Sigma_2(\lambda,B) = {TRUE,FALSE} )</td>
<td></td>
</tr>
</tbody>
</table>

Now, we have enough machinery to use algebraic concepts for formulating a mechanism for performing language translation. We will use \( \Pi_1 \) and \( \Pi_2 \) to define a tree transducer, which will transform syntax trees of one grammar into syntax trees of another grammar. The representation functions enable us formally to associate the source and target trees in a row of a tree transducer, which will be defined in Chapter 4. For instance, \( \Pi_1(c) \) and \( \Pi_2(c) \) may associate a source tree and target tree in the definition of a tree transducer. Figure 2.6 indicates one row of a tree transducer, and the association of arguments shown as arrows is automatically specified by \( \Pi_1(c) \) and \( \Pi_2(c) \).
Figure 2.6 A row of a tree transducer
Previously, we stated that if $A$ and $B$ are $\Sigma$-algebras, and $\overline{A}$ and $\overline{B}$ are derived $\Sigma$-algebras, then $h: A \rightarrow B$ is a homomorphism if and only if $\overline{h} = h(u_{\lambda}: \emptyset \rightarrow \emptyset)$ is a homomorphism from $\overline{A}$ to $\overline{B}$. Now similarly we state the following theorem.

**Theorem 2.21**

Let $\Sigma'$ be an $N'$-sorted signature, which is representable in an $N$-sorted signature of $\Sigma$. Let $A'$ and $B'$ be $\Sigma'$-algebras represented in $\Sigma$-algebras $A$ and $B$ respectively. If $h: A \rightarrow B$ is a homomorphism of $\Sigma$-algebras, then $h' = (h_{\pi}/\eta(n') = \eta$ for $n' \in N')$ is a homomorphism of $\Sigma$-algebras from $A'$ to $B'$.

**Proof:** Let $\pi$ and $\eta$ be representation functions, then consider any $f' \in \Sigma'(a', n')$ with $f \in \Sigma(\alpha, n)$ where $f = \pi(f')$, $n = \eta(n')$, and $\alpha = \eta(\alpha')$.

Then $h'_{\eta}(f'_{A'}(x)) = h_{\eta}(\pi(f')_{A}(x))$

$$= h_{\eta}(f_{A}(x))$$
$$= f_{B}(h_{\eta}(\alpha)(x)) \text{ since } h: A \rightarrow B \text{ is a homomorphism}$$
$$= \pi(f')_{B}(h_{\eta}(\alpha')(x))$$
$$= f'_{B}(h_{\eta}(\alpha')(x))$$

Hence $h': A' \rightarrow B'$ is a homomorphism of $\Sigma'$-algebras. $\square$

Theorem 2.21 shows that if $h: A \rightarrow B$ is a homomorphism of $\Sigma$-algebras, and if $\Sigma'$ is representable in host signature of $\Sigma$, then there is a represented homomorphism $h': A' \rightarrow B'$. Consider for instance an LDS $(G, A)$ with signature of $\Sigma$ and a semantic homomorphism $\mu: T_{\Sigma} \rightarrow A$. As discussed earlier, $\mu$ induces a "derived" homomorphism $\overline{\mu}: T_{\overline{\Sigma}} \rightarrow \overline{A}$. Thus,
if signature $\Sigma'$ were representable in host signature $\bar{\Sigma}$, then $\bar{\mu}': \bar{T}_{\bar{\Sigma}} \to \bar{A}'$ would be a homomorphism of $\Sigma'$-algebras. This notion has been found useful in correctness of tree transducers, which will be discussed in Chapter 4.

Formally, in Chapter 4, we will see how the notions of derived and represented signatures and homomorphisms between algebras can be used to define tree transducers. These will then construct the output (transduced tree) associated with a given input (source tree) by considering the source tree to be composed of derived operations.

In summary, we have so far presented development of the languages by an algebraic interpretation.
3. REDUCTION ON DATA STRUCTURE TREES

Data structures will be specified, as we shall see by a tuple, one of whose components is a set of "equations''. In this chapter, we investigate some consequences of applying a set of "equations'' $E$ to a set of trees (operations of $\mathcal{E}$), which are associated with the derived signature of $\Sigma$ of a data structure. A formalism needed to be developed for studying application of $E$ to $f$ in $\mathcal{E}$. One of our aims in this chapter is to establish certain facts about "reduced trees,'' trees to which equations are no longer applicable. Later on, we will use the results of this chapter to develop the background for natural construction of tree transducers for an implementation of a source data structure $d_1$ by target data structure $d_2$.

The theory to be developed here centers on determining a reduced form of a tree $f$ in $\mathcal{E}$ when equations can be no longer applied to it. This development depends on some lattice theory, which will be reviewed.

3.1. Algebraic Equations

This section discusses the "equations'', which can be used in expressing, simplifying, and implementation of data structures. In the algebraic framework, we will consider a data structure as a signature along with a set of equations $E$. Hoare [17] and Guttag [13]
and some other researchers defend the notion of a data structure as a set of values together with a set of primitive operations on those values.

In this section, we will study "equations" which not only can be used for specifying the properties of data structures but also can be used to simplify the data structure elements.

Guttag [13] and Goguen [11] have used algebraic equations for an implementation of a data structure. Related work has been done by Hoare [17]. Goguen [9, 11] has investigated that algebraic equations can be used not only to define more complex data structures, but also to define the meaning of data structures.

The following example illustrates the concept of "algebraic equations".

Example 3.1

In order to illustrate the concept of "algebraic equations"; we employ the example of Linear Lists (S-expression data structures of atoms from a set A). In [2], they have used Pascal programming language for an implementation of S-expressions, which form the basis for pure Lisp. The set of S-expressions is defined recursively as the smallest set such that

i) Atoms are S-expressions (Atoms are members of a scalar type)

ii) If S1 and S2 are S-expressions, then so is \((S1 \cdot S2)\)

In order to manipulate S-expressions, we consider the following operations (functions), which are defined below:
1) CAR, CDR are selector functions. CAR takes an S-expression (S1·S2) and returns S1. CDR takes an S-expression (S1·S2) and returns S2.

2) EQUAL is intended to test equality of two S-expressions.

3) CONS is a function which takes two S-expressions S1 and S2 and returns (S1·S2).

4) NIL is a constant function which denotes the empty S-expression.

For example, if t=(A·(B·C)) is an S-expression then

\[
\begin{align*}
\text{CAR}(t) &= A \\
\text{CDR}(t) &= (B\cdot C) \\
\text{CONS}(t, t) &= ((A\cdot (B\cdot C))\cdot (A\cdot (B\cdot C)))
\end{align*}
\]

First, we define \( \Sigma \) as an N-sorted signature for specification of S-expressions and then we provide the equations, which are specifying the properties of S-expressions data structure.

\( N = \{ S, B, A \} \) where S, B, and D stand for S-expressions, Boolean, and Atoms respectively.

Signature \( \Sigma \):

\[
\begin{align*}
\Sigma(S, S, S) &= \{ \text{CONS} \} \\
\Sigma(\lambda, S) &= \{ \text{NIL}, \text{ERROR} \} \\
\Sigma(S, S) &= \{ \text{CAR}, \text{CDR} \} \\
\Sigma(S, B) &= \{ \text{EQUAL} \} \\
\Sigma(B, B) &= \{ \text{AND} \}
\end{align*}
\]
Informally, \( \text{CAR}((\text{CONS}(x_1, x_2))) = x_1 \) specifies the meaning of the CAR function and \( \text{CDR}((\text{CONS} (x_1, x_2))) = x_2 \) specifies the meaning of the CDR function. The above two equations can be written formally as:

\[
\begin{align*}
\text{e1: } \text{CAR}([\text{CONS}[X_1, X_2]]) &= x_1 \\
\text{e2: } \text{CDR}([\text{CONS}[X_1, X_2]]) &= x_2
\end{align*}
\]

Equation (e1) indicates that the CAR is a function which selects the first component of a list, while equation (e2) indicates that CDR is a function, which selects the second component of a list. In order to clarify the equations, we will not write the domain of equations.

Thus, equations (e1) and (e2) will look like:

\[
\begin{align*}
\text{e1: } \text{CAR}([\text{CONS}[X_1, X_2]]) &= X_1 \\
\text{e2: } \text{CDR}([\text{CONS}[X_1, X_2]]) &= X_2
\end{align*}
\]

The rest of the equations can be written as:

\[
\begin{align*}
\text{e3: } \text{CAR}([\text{NIL}]) &= \text{ERROR} \\
\text{e4: } \text{CDR}([\text{NIL}]) &= \text{ERROR} \\
\text{e5: } \text{EQUAL}([\text{CONS}[X_1, X_2], [\text{NIL}]) &= \text{FALSE} \\
\text{e6: } \text{EQUAL}([\text{NIL}, [\text{CONS}[X_1, X_2]]) &= \text{FALSE} \\
\text{e7: } \text{EQUAL}[X_1, X_1] &= \text{TRUE} \\
\text{e8: } \text{EQUAL}([\text{CONS}[X_1, X_2], [\text{CONS}[X_3, X_4]]) &= \text{AND}([\text{EQUAL}[X_1, X_3], \text{EQUAL}[X_2, X_4]]) \\
\text{e9: } \text{CONS}([\text{NIL}, X_1]) &= X_1 \\
\text{e10: } \text{CONS}([X_1, [\text{NIL}]) &= X_1
\end{align*}
\]

For instance, equation e8 indicates that two lists \( \text{CONS}(x_1, x_2) \) and \( \text{CONS}(x_3, x_4) \) are equal if and only if \( x_1 = x_3 \) and \( x_2 = x_4 \). \( \square \)
The preceding example introduced an informal concept of semantic equations (equations). Now, we give the formal definition of equations, which will be used in implementation of data structures. Later on, we will generalize equations to a class of logical sentences called "Horn clauses" (conditional equations). For the moment we use the terms "equations", "simple equations", and "semantic equations" interchangeably.

**Definition 3.2 [Simple Equation]**

Let \( \Sigma \) be an \( N \)-sorted signature. Then a set of "simple equations" is an indexed family of sets

\[
E = \{ E_n / n \in N \text{ and } \langle L, R \rangle \in E_n \text{ implies that } L, R \in \Xi(w, n) \text{ for some } w \in N^* \}
\]

Furthermore, if \( A \) is a \( \Sigma \)-algebra, then \( A \) satisfies \( E \) if and only if for each \( \langle L, R \rangle \in E_n \), \( L_A = R_A \) (i.e., the functions \( L_A \) and \( R_A \) are equal).

Finally, we will often write an element \( \langle L, R \rangle \in E \) as an "equation" \( L = R. \)

We reserve the symbols \( L, L', L'', ..., R, R', R'', ... \) for equations. Informally, \( L = R \) means that the sequence of operations expressed by the trees \( L \) and \( R \) have the same behavior. That is, when values are substituted for variables in \( L \) and \( R \), the instantiated trees interpret to equivalent values. The term "simple equations" or "equations" will be used instead of "semantic equations" in the following chapters.
3.2. Informal Description of a Rewrite Rule

Informally, the set of "reduced trees" of a tree $t$ under $E$ (a set of equations) is a set of trees $t'$ such that further application of equations of $E$ to $t''$'s will not be useful (they yield members of the set of reduced trees of $t$). Typically, when a user specifies an equation for a data structure, the intent is as a "simplification rule". For instance,

$$\text{addtoset}(\text{addtoset}(s,x), x) = \text{addtoset}(s,x)$$

says that the composite operation of adding $x$ to the same set twice may be replaced by the operation of adding it once. Based on this operation, we deal with equations as "directed rewriting rules", specifying that the lefthand side of equation is to be rewritten by the righthand side.

Huet and Oppen [23] refer to such equations as "directed equations", and use them to define "term rewriting systems". Clearly, equations in the classical sense are easily dealt within this framework by expressing each equation as two rewriting rules, one replacing the lefthand side by the righthand side, and another replacing the righthand side by the lefthand side. Furthermore, this directional view of equation gives us a notion of "reduction" where one tree $t'$ is "more reduced" than $t$ if $t'$ is obtained from $t$ by applying a directional equation to it. This, consequently, yields the set of reduced forms of a tree, which is a set of trees that cannot be
further reduced by application of equations. We, thus, often use the phrase "tree rewriting system" instead of "equations".

3.3. Previous and Related Work

The idea of a "rewrite rule" has been investigated by several researchers, including Guttag et al. [13, 16], Huet et al. [22], Goguen et al. [11], Rosen [38], and Knuth and Bendix [27] to mention a few. Our development is similar to that of Guttag et al. [15], who investigated conditions to be imposed on rewrite rules to guarantee the termination of a rewriting system. In [22] and [27], they have proved the undecidability of termination of rewriting systems and in [16], Guttag has imposed syntactically checkable conditions, which provides a formal basis for termination of some term (tree) rewriting systems.

Knuth and Bendix [27] have investigated a partial ordering on trees such that if the lefthand side of an equation (rewrite rule) was greater than the righthand side in their ordering, then the equation could be constructed as a reduction. Rosen [38] has discussed Church-Rosser theorems for rewriting systems, which address the issue of normal forms in tree rewriting systems.

In this work, we construct a "Lattice", given a set \( \Theta \) of trees (derived operations of the signature \( \Sigma \)), and a tree rewriting system (set of equations) that is applicable to \( \Theta \); we are not primarily concerned with the termination or otherwise of the tree rewriting systems, though a set of conditions for termination will be presented
later. Our goal is to eventually partition \( \Theta \) into classes of trees that are "equivalent" under the tree rewriting system we provide. The Lattice will be "complete", and the "least element" of this lattice will be most useful for our purposes (section 3.6.3). We will, however, use the techniques of [16] to make the construction of this Lattice algorithmic.

3.4. Application of a Rewrite Rule to a Tree

The formalization of the concept of application of \( E \) to \( \Sigma \) is given below.

**Definition 3.3** [Application of \( E \) to \( \Sigma \)]

Let \( \Sigma \) be an \( N \)-sorted signature and let \( E = \{ e_1, \ldots, e, \ldots, e_n \} \) be a set of simple equations. Let \( <L,R> \) be an equation \( e \in \Sigma(w,s) \).

Then \( \Rightarrow \) is a relation defined as:

(i) if \( f = L, f' = R \) then \( f \Rightarrow f' \).

(ii) if \( f = L_{\Sigma}[f_1, \ldots, f_m] \) and \( f' = R_{\Sigma}[f_1, \ldots, f_m] \) then

\[
f \Rightarrow f'
\]

(iii) if \( f = g_{\Sigma}[f_1, \ldots, f_j, \ldots, f_m], \ f' = g_{\Sigma}[f_1, \ldots, f_j, \ldots, f_m], \text{ and } f \Rightarrow f' \text{ then } f \Rightarrow f'.
\]

Furthermore, \( f \Rightarrow e_f f' \). Also, we define \( f_E \) as \( f_E = \{ f' / f \Rightarrow f' \}, e \in E \). If \( f_1 \Rightarrow f_1 \) then we write \( f_E \Rightarrow f_1 \) for some \( f \) in \( \Sigma \). The relation \( \Rightarrow \) will be called a "reduction relation".

We denote the reflexive-transitive and transitive-closure of \( \Rightarrow \) by \( \Rightarrow^* \) and \( \Rightarrow \) respectively. The symbols \( \not\Rightarrow \), \( \not\Rightarrow \), and \( \not\Rightarrow \) denote the com-
plement of $\Rightarrow_\mathcal{E}$, $\Rightarrow^*$, and $\Rightarrow^{+}$ respectively, i.e., if $f_1 \Rightarrow_\mathcal{E} f_2$ then it is impossible to obtain $f_2$ by applying equations of $\mathcal{E}$ to $f_1$. The subscript $\mathcal{E}$ will be left out when understood from the context. From now on, we will use and consider equations $\mathcal{E}$ as "rewrite rules", and the terms "simple equations", "semantic equations", and "tree rewriting system" will be used for the same purpose.

Directed rewriting rules such as $\mathcal{E}$ have received much interest [22, 38]. One of the key issues of such rewriting rules is the determination of "normal forms", and the notion of equivalences of "normal forms". The question essentially posed is, given trees $t_1$ and $t_2$ and a set of directed equations $\mathcal{E}$, can we determine that $t_1$ and $t_2$ are "equivalent" by reducing them using $\mathcal{E}$ to a common "normal form"? Properties of this nature have been referred to as Church-Rosser properties, due to the initial investigations of this problem in the Lambda calculus [5]. While our work does not directly deal with the issues of normal forms and Church-Rosser theorems, we briefly present some of this work in our context below.

To begin with, we need to define the set of "normal forms" of a tree in $\tilde{\Sigma}$. Following this, we will present Church-Rosser properties and an example.

**Definition 3.4** $[f_o \Rightarrow^E f_n]$

Let $E = \{e_1, \ldots, e_m\}$ be a set of equations. If

$f_o \Rightarrow e_1 \Rightarrow e_2 \Rightarrow \ldots \Rightarrow e_n$ and $e_i \in E$, $i=1, \ldots, n$ then we write
\( f_0 \xrightarrow{E} f_n \) or \( f_0 \xrightarrow{E^n} f_n \). If \( E \) is understood from the context then the subscript \( E \) will be left out.

**Definition 3.5** [Normal Form of a Tree]

Let \( f_0, f_1, \ldots, f_n \) be in \( \Sigma \) and \( E = \{ e_1, \ldots, e_m \} \). If \( f_0 \Rightarrow f_1 \Rightarrow \ldots \Rightarrow f_n \) and if there does not exist any \( f' \) in \( \Sigma \) such that \( f_n \Rightarrow f' \) for any \( e \) in \( E \), then \( f_n \) will be considered a "normal form" of \( f_0 \). Furthermore, for \( f \) in \( \Sigma \), \( NORM(f) \) is the set of all normal forms of \( f \).

If \( NORM(f) \) is a singleton set for each \( f \) in \( \Sigma \), then it means that every element of \( \Sigma \) has a unique normal form. We, next define a Church-Rosser set of rewrite rules and related concepts.

**Definition 3.6** [Church-Rosser]

A set of equations \( E \) is said to be Church-Rosser if and only if \( t, t_1, t_2 \in \Sigma, t \xrightarrow{*} t_1 \) and \( t \xrightarrow{*} t_2 \) implies that there exist \( t_3 \) in \( \Sigma \) such that \( t_1 \xrightarrow{*} t_3 \) and \( t_2 \xrightarrow{*} t_3 \).

The Church-Rosser property pictorially is represented as in Figure 3.1. The use of Church-Rosser properties for issues like "tree manipulation systems" was investigated by Rosen [38].

The Church-Rosser property guarantees that any two reduction sequences starting with a tree \( t \) which end in normal forms, end in the same normal form. We should note that \( t \) may have a normal form and also an infinite sequence of reduction. In order to find a normal form whenever such exists, we have to choose a reduction se-
Figure 3.1 Church-Rosser property
quence carefully. If for each tree \( p \), there is a constant \( k_p \), such that, no more than \( k_p \) successive applications of the transformation (applying \( E \) to \( p \)) are possible, starting with \( p \) and if a transformation is iteratively applied to \( p \) until no longer possible, a unique tree \( p \) is reached then we call such a transformation \( (\Rightarrow) \) Finite Church-Rosser (FCR). Formally, we define FCR as below.

**Definition 3.7 [Finite Church-Rosser]**

We say \( \Rightarrow \) is Finite Church-Rosser (FCR) if

1. for each \( p \) in \( T_\Sigma \), there is a constant \( k_p \) such that if \( p \xrightarrow{E} q \), then \( n < k_p \), and
2. \( q_1, q_2 \in \text{NORM}(P) \) implies \( q_1 = q_2 \).

We state the following theorem from [1] without providing the proof.

**Theorem 3.8**

Let \( \Rightarrow \) be a relation on \( T_\Sigma \), then \( \Rightarrow \) is FCR if and only if the following conditions are satisfied.

1. for each \( t \) in \( T_\Sigma \), there is a constant \( k_t \) such that if \( t \xrightarrow{E} t' \), then \( n < k_t \), and
2. for all \( t \) in \( T_\Sigma \), if \( t \xrightarrow{E} t_1 \) and \( t \xrightarrow{E} t_2 \), then there is some \( t_3 \) such that \( t_1 \xrightarrow{E} t_3 \) and \( t_2 \xrightarrow{E} t_3 \).

The following example illustrate the Church-Rosser property for stack data structure. That is, there is a normal form for all \( t \) in \( T_\Sigma \).

**Example 3.9**

Let \( N = \{\text{Stack, Data, Boolean}\} = \{S, D, B\} \) be a set of sorts and \( \Sigma \) be an \( N \)-sorted signature as:
\[ \Sigma(\lambda, D) = \{1, 2, 3, 4, 5, \text{UNDEFINED} \} \]
\[ \Sigma(\lambda, S) = \{\text{NEWSTACK, UNDERFLOW} \} \]
\[ \Sigma(SD, S) = \{\text{PUSH} \} \]
\[ \Sigma(S, S) = \{\text{POP} \} \]
\[ \Sigma(S, B) = \{\text{ISEMPTY} \} \]
\[ \Sigma(S, D) = \{\text{TOP} \} \]

Also, let \( E = \{e_1, e_2, e_3, e_4, e_5, e_6\} \) such that

- \( e_1: \text{POP[NEWSTACK]} = \text{UNDERFLOW} \)
- \( e_2: \text{POP[PUSH}[X_1X_2]] = X_1 \)
- \( e_3: \text{ISEMPTY[NEWSTACK]} = \text{TRUE} \)
- \( e_4: \text{ISEMPTY[PUSH}[X_1X_2]] = \text{FALSE} \)
- \( e_5: \text{TOP[PUSH}[X_1X_2]] = X_2 \)
- \( e_6: \text{TOP[NEWSTACK]} = \text{UNDEFINED} \)

Now let

\[ t = \text{POP}<\text{PUSH}<\text{PUSH}<\text{PUSH}<\text{NEWSTACK 3}>>2>5>> \]

then

\[ t \xrightarrow{e_2} t_1 \xrightarrow{e_2} t_3 \text{ and } t \xrightarrow{e_2} t_2 \xrightarrow{e_2} t_3 \]

where

\[ t_1 = \text{POP}<\text{PUSH}<\text{PUSH}<\text{NEWSTACK 2}>>5>>, \]
\[ t_2 = \text{PUSH}<\text{POP}<\text{PUSH}<\text{NEWSTACK 3}>>2>>, \]
\[ t_3 = \text{PUSH}<\text{NEWSTACK 2}>> \]

\( t_3 \) is the normal form of \( t \) and it is unique. Thus, the Church-Rosser property is automatically satisfied. Since the order of elements is important in stack data structure, it can be proved that there
is a unique normal form for all $t$ in $T_\Sigma$. It will be shown that this is not true for **set** data structure to be described shortly. □

Our work does not directly deal with Church-Rosser properties in general. This is mentioned in this context because of its relevance to this research. We are, however, concerned with the notion of "reduced trees", which are defined after a brief discussion of the preceding example.

We should note that it is possible to include the REPLACE operation for **stack** data structure. That is, we may add $\Sigma(SD,S) = \{\text{REPLACE}\}$ to $\Sigma$ such that REPLACE$(s,d)$ replaces the topmost element of stack $s$ with data value of $d$. Thus, informally we can write

$$e9: \text{REPLACE}(S,d) = \text{PUSH}(\text{POP}(S),d)$$

The equation (e9) is not completely well defined because if stack $s$ is empty the POP$(s)$ will cause an underflow. In Chapter 5, we will remove this drawback and introduce a more powerful class of equations (known as Horn clauses or conditional equations) that can handle these cases.

### 3.5. Reduced Trees under Rewriting Systems

This section introduces the definition of the "cycle set" of a tree $f$ in $\Sigma$ and the "reduction" of a tree $f$ in $\Sigma$ under tree rewriting systems. These definitions are crucial in our developments for the following chapters. Basically, if $f \in \Sigma$, then the cycle set of $f$ is the set of trees, which makes a circle. That is, if $f \in \Sigma$ and
\[ f \Rightarrow f_1 \Rightarrow f_2 \ldots \Rightarrow f_n \] and \( f_n \Rightarrow f \) then \( \{f_1, \ldots, f_n\} \) is the cycle set of \( f \). There may exist other circles for a given \( f \), but we will select an appropriate one.

Below, the definition of "cycle set of a tree" and related concepts are given.

**Definition 3.10** [Cycle Set of a Tree]

Let \( f \) be in \( \Sigma \) and \( E \) be a set of equations associated with \( \Sigma \). The "cycle set of \( f \) under \( E \)" is the set \( \{g/f \Rightarrow g \mid g \Rightarrow f\} \) and is designated by \( [f]_E \). Note that if for every \( g \) such that \( f \Rightarrow g \) we have \( g \not \Rightarrow f \), then \( [f]_E = \emptyset \). The cycle set of \( f \) (i.e., \( [f]_E \)) is said to be "final" if \( g \in [f]_E \) and \( g \Rightarrow h \) implies \( h \in [f]_E \). Also, we will use the phrase "final cycle set of \( f \)" to denote that the cycle set of \( f \) is final. The subscript \( E \) will be left out when understood from the context.

**Definition 3.11** [Irreducible Tree]

Let \( f \) be a tree in \( \Sigma \). Then \( f \) under \( E \) is called "irreducible" if there does not exist any \( e \in E \) and \( g \in \Sigma \) such that \( f \Rightarrow g \).

**Definition 3.12** [Nonreducible Forms of a Tree]

Nonreducible forms of a tree \( f \) (NONRED(\( f \))) is the set of trees which are derived from \( f \) and are irreducible. That is, \( \text{NONRED}(f) = \{g/f \Rightarrow g \text{ and } g \text{ is irreducible}\} \).

**Theorem 3.13**

If \( f \) is irreducible then \( \text{NONRED}(f) = \{f\} \).
Let $f$ be in $\mathbb{E}$. Then informally, the reduction of $f$ is the union of two sets, namely $\text{NONRED}(f)$ and \{f'/$f^*_{E}f''$ where $f'e[f'']_E$ and $[f'']_E$ is final\}. For example, if $f\Rightarrow g\Rightarrow f'_0$, $[f'_0] = \{f'_0, f'_1, f'_2, f'_3, f'_4\}$, and $f\Rightarrow h\Rightarrow f''_0$ and $[f''_0] = \{f''_0, f''_1\}$ and $f\Rightarrow f_1\Rightarrow f_2\Rightarrow f_3$ and $f_3$ is irreducible. Then the reduction of $f$ (i.e., $[[f]]$) may be pictured as in Figure 3.2.

Before introducing the definition of reduction of a tree in $\mathbb{E}$ under $E$, we exemplify the above concepts by the following examples.

Example 3.14

Let $N = \{\text{Set, Integer}\} = \{S, I\}$ be a set of sorts and $\Sigma$ be an N-sorted signature:

$\Sigma(\lambda, I) = \{a, b, c, \ldots\}$

$\Sigma(IS, S) = \{F\}$

$\Sigma(\lambda, S) = \{\emptyset\}$

Also, let $E = \{e_1, e_2\}$ where (for the sake of clarity, we drop out the domains of equations)

$e_1: F[X_1F[X_2X_3]] = F[X_2F[X_1X_3]]$

$e_2: F[X_1F[X_1\emptyset]] = F[X_1\emptyset]$

Intuitively, $F(x,y)$ is the operation to add the value of $x$ to the set $y$. Equation $e_1$ states that the order of elements in a set is unimportant, and equation $e_2$ states that an element need be added only once.

Let $t = F<aF<bF<aF<b\emptyset>\ldots$. The following reductions will clarify that why we are interested in selecting the "final" cycle set.
\[ t \equiv t_1 \equiv t_2 \equiv t_3 \equiv t_4 \equiv t_5 \equiv t_6 \quad (\star 1) \]
\[ t_6 \equiv t_7 \equiv t_8 \equiv t_7 \quad (\star 2) \]
\[ t \xrightarrow{\star} t_4 \equiv t_5 \equiv t_4 \quad (\star 3) \]
\[ t \xrightarrow{\star} t_5 \equiv t_6 \equiv t_5 \quad (\star 4) \]

where

\[ t_1 = F < bF < aF < aF > b0 > > > > > \]
\[ t_2 = F < bF < aF < bF > a0 > > > > > \]
\[ t_3 = F < bF < bF < aF > a0 > > > > > \]
\[ t_4 = F < bF < bF < aF > b0 > > > > > \]
\[ t_5 = F < bF < aF < b0 > > > > > \]
\[ t_6 = F < aF < bF > b0 > > > > > \]
\[ t_7 = F < aF < b0 > > > > > \]
\[ t_8 = F < bF < a0 > > > > > \]

Of course, there exist many other sequences of reductions. Figure 3.3 is the pictorial representation of the reductions \((\star 1)\), \((\star 2)\), \((\star 3)\), and \((\star 4)\). The above reductions imply that \([t_7] = \{t_7, t_8\}\), \([t_4] = \{t_4, t_5\}\), \([t_5] = \{t_5, t_6\}\).

Thus, we have

\[ t \xrightarrow{\star} t_7 \Rightarrow t_8 \Rightarrow t_7 \]
\[ t \xrightarrow{\star} t_4 \Rightarrow t_5 \Rightarrow t_4 \]
\[ t \xrightarrow{\star} t_5 \Rightarrow t_6 \Rightarrow t_5 \]

The above reductions state that there are several candidates for reduction set of \(t ([t])\); among those are \([t_7]\), \([t_4]\), and \([t_5]\).
Figure 3.2 Pictorial representation of $[[f]]$
Figure 3.3 Pictorial representation of $[t]$
By examining those equivalence sets, we notice that only \([t_7]\) is a "final" equivalence set, and the others are not final because if 
\[t_7 \Rightarrow g_1 \text{ or } t_8 \Rightarrow g_2\] then \(g_1 \cdot g_2 \in [t_7]\) but if \(t_4 \Rightarrow h\) then \(h \not\in [t_4]\).

Thus, \([t_7]\) is the reduction of \(t\) under \(E\).

Before studying the formal definition of the reduction of \(f\) (i.e., \([[f]]\)), in \(\Sigma\) and in order to exemplify the concept of \([[f]]\), we examine the following example.

Example 3.15

Let \(E = \{e_1, e_2, e_3, e_4\}\) be the following equations associated with \(\Sigma\). Also let \(f\) be in \(\Sigma\) such that \(f = a[b[X_1]c[X_2]]\).

\[
\begin{align*}
e_1: & \quad a[X_1c[X_2]] = d'[X_1X_2] \\
e_2: & \quad a[b[X_1]X_2] = d''[X_1X_2] \\
e_3: & \quad d'[X_1X_2] = d'[X_2X_1] \\
e_4: & \quad d''[X_1X_2] = d''[X_2X_1]
\end{align*}
\]

Using

\[
\begin{align*}
f_1' &= d'[b[X_1]X_2] \quad f_2' = d'[X_2b[X_1]] \quad f_1'' = d''[X_1c[X_2]] \quad f_2'' = d''[c[X_2]X_1]
\end{align*}
\]

and applying \(E\) to \(f\) will yield the following reductions.

1. \(f \xrightarrow{\bar{e}_1} f_1' \xrightarrow{\bar{e}_3} f_2' \xrightarrow{\bar{e}_3} f_1'\)
2. \(f \xrightarrow{\bar{e}_2} f_1'' \xrightarrow{\bar{e}_4} f_2'' \xrightarrow{\bar{e}_4} f_1''\)

Thus, (1) implies that reduction of \(f\) should be \(\{f_1', f_2'\}\) and (2) implies that then should be \(\{f_1'', f_2''\}\). As we see \(\{f_1', f_2'\}\) and \(\{f_1'', f_2''\}\) are both in the reductions of \(f\) under \(E\), because both are final cycle sets. In the first glance, it may seem that \(\{f_1', f_2'\}\) and \(\{f_1'', f_2''\}\) are not equivalent (the same meaning) because they cannot
be reduced to each other and besides they have different representation forms. But the matter of fact is that, definition of $E$ guarantees that all of $f_1', f_2', f_1'' f_2''$ have the same meaning. So, we may consider $\{f_1', f_2', f_1'' f_2''\}$ as the reduction of $f$ under $E$ remembering the fact that $\{f_1', f_2\}$ and $\{f_1'' f_2''\}$ are "final" cycle sets. 

**Definition 3.16** [Reduction of a Tree]

Given a nonterminated tree $f$ in $\Sigma$ and a set of equations $E$ associated with $\Sigma$, we define the "reduction of $f$ under $E$" (i.e., $[[f]]_E$) to be the set $[[f]]_E = \text{NONRED}(f) \cup \{f'/f_E \geq f''$ and $f' \in [f'']_E$ and $[f'']_E$ is final\}. We call $[[f]]_E$ the set of "reduced trees" of $f$ under $E$, or the set of reduction of $f$ under $E$. Furthermore, if $\emptyset = \{f_1, \ldots, f_n\}$, $f_i \in \Sigma$ for $i = 1, \ldots, n$, then $[[\emptyset]]_E = \cap_{i=1}^n [[f_i]]_E$. The subscript $E$ will be left out when understood from the context. 

Thus, $x \in [[f]]$ immediately implies that $f \Rightarrow^* x$; and if $f \in [[f]]$, it means that $f$ itself is in the reduced form and cannot be reduced further.

The concept of the reduction of $f$ will be used for construction of a lattice over a set of trees associated with a data structure (signature $\Sigma$ along with equations $E$). An important point to note is that for some data structures such as set data structures, $[[f]]$ is not often a unique element (singleton set). Instead, we have many reduced forms for a given tree. This is because of the form of rewriting rules (equations). For instance, the trees $t_7$ and $t_8$ (in example 3.14) have the same meaning (the set $\{a, b\}$) but are distinct trees.
We next state and prove theorems and Lemmas regarding the reduction of trees.

**Theorem 3.17**

If $f$ is irreducible the $[[f]] = \{f\}$.

**Proof:** From Theorem 3.14, $\text{NONRED}(f) = \{f\}$.

Also, $f^* = f^*$ implies $f^* = f$ since $f$ is irreducible.

Then $[f^*] = [f]$

$$= \{g/f^* = g = f\}$$

$$= \emptyset$$ since $f^* = g$ for any $g$ since $f$ is irreducible.

Thus $[[f]] = \{f\} \cup \emptyset = \{f\}$. \hfill $\square$

**Lemma 3.18**

Let $E$ be a set of simple equations associated with $E$. Let $t_i$ be in $\Sigma(\lambda, s_i)$ for $i=1,\ldots,n$ and $w=s_1\ldots s_n$; and $P,q\in E(w,s)$. If $[[P]]_E = [[q]]_E$, then

$$[[P_{T_\Sigma(t_1,\ldots,t_n)}]]_E = [[q_{T_\Sigma(t_1,\ldots,t_n)}]]_E.$$ 

**Proof:** Case (1): $P,q\in E(\lambda,s)$, then the lemma is trivially true.

Case (2): $P,q\in E(w,s)$, then property $[[P]]_E = [[q]]_E$ implies that $P^* \in X$ and $q^* \in X$ where $X \in [[P]]_E$. Then

$$P_{T_\Sigma(t_1,\ldots,t_n)}^* \in X_{T_\Sigma}(t_1,\ldots,t_n);$$ and

$$q_{T_\Sigma(t_1,\ldots,t_n)}^* \in X_{T_\Sigma}(t_1,\ldots,t_n),$$ which implies that

$$[[P_{T_\Sigma(t_1,\ldots,t_n)}]]_E = [[q_{T_\Sigma(t_1,\ldots,t_n)}]]_E.$$ 

Assuming $P$ and $q$ have only one cycle set, Figure 3.4 is the pictorial interpretation of Lemma 3.18. \hfill $\square$
Figure 3.4. Pictorial representation of Lemma 3.18
Lemma 3.19

Let \( E = \{ e_1, \ldots, e_m \} \) be a set of simple equations associated with \( \Sigma \). If \([[P]]_E = [[P']]_E \) and \([[t_i]]_E = [[t'_i]]_E \) for \( i = 1, \ldots, n \) then \([[P T (t_1, \ldots, t_n)]_E = [[P' T (t'_1, \ldots, t'_n)]_E \). 

Proof: Let \( q_i \in [[t_i]]_E \), then \( q_i \in [[t'_i]]_E \) since \([[t_i]]_E = [[t'_i]]_E \) for \( i = 1, \ldots, n \).

Then \([[P T (t_1, \ldots, t_n)]_E = [[P T (q_1, \ldots, q_n)]_E \) since \( q_i \in [[t_i]]_E \) from Lemma 3.18

\[ = [[P' T (q_1, \ldots, q_n)]_E \]

\[ = [[P' T (t'_1, \ldots, t'_n)]_E \) since \( q_i \in [[t'_i]]_E \). \]

3.6 A Lattice Theoretic Approach to Equivalences in Tree Rewriting Systems

We now review some lattice theory needed for our development. We first define the term "partially ordered set" and the concept of a "lattice" [12, 43]. Then the notion of a "complete lattice" will be shown for a set of trees under tree rewriting system.

3.6.1. Lattice theory background

Definition 3.20 [Partially Ordered Set]

A "partially ordered set" (poset) is a pair \((L, \leq)\) where \( L \) is a nonempty set and \( \leq \) is a binary relation on \( L \) satisfying the following properties: for all \( b_1, b_2, b_3 \in L \):

1. \( b_1 \leq b_1 \) (reflexive).
2. \( b_1 \leq b_2 \) and \( b_2 \leq b_1 \) implies \( b_1 = b_2 \) (antisymmetric).
3. \( b_1 \leq b_2 \) and \( b_2 \leq b_3 \) implies \( b_1 \leq b_3 \) (transitive).
Definition 3.21 [Least Upper Bound, Greatest Lower Bound]

Let \((L, \leq)\) be a poset, and let \(b_1\) and \(b_2\) be elements in \(L\). An element \(a \in L\) is called a "greatest lower bound" (GLB) of \(b_1\) and \(b_2\), if \(b_1 \leq a\), \(b_2 \leq a\), and if for any \(a' \in L\), \(b_1 \leq a'\) and \(b_2 \leq a'\) imply \(a \leq a'\); and \(a\) is designated by \(a = \text{GLB}(b_1, b_2)\). Similarly, an element \(c \in L\) is called a "least upper bound" (LUB) of \(b_1\) and \(b_2\), if \(c \geq b_1\), \(c \geq b_2\), and if for any \(c' \in L\), \(c' \geq b_1\) and \(c' \geq b_2\) imply \(c' \geq c\); and \(c\) is designated by \(c = \text{LUB}(b_1, b_2)\). □

Definition 3.22 [Lattice, Complete Lattice]

A "lattice" is a poset \((L, \leq)\) in which every pair of elements \(\pi_1, \pi_2 \in L\) has a GLB and a LUB. A lattice is called "complete" if \(\text{LUB}(H)\) and \(\text{GLB}(H)\) exist for all \(H \subseteq L\). □

We next define a binary relation \(\geq\) between components of a poset determined by \(\mathcal{E}\).

Definition 3.23 [Binary Relation]

Let \(\Theta = \{f_1, \ldots, f_n\}\), \(f_i \in \mathcal{E}\) for \(i = 1, \ldots, n\), and \(P(\Theta)\) be the set of all nonempty partitions of \(\Theta\). For all \(\pi_1, \pi_2 \in P(\Theta)\), \(\pi_1 \geq \pi_2\) if and only if every \(\chi \in \pi_2\) is a subset of some \(\gamma \in \pi_1\). □

It is left to the reader to verify that \(P(\Theta)\) under binary relation \(\geq\) forms a poset.

Example 3.24

Let \(\Theta = \{f_1, f_2, f_3\}\). Then 
\[
P(\Theta) = \{\{f_1, f_2, f_3\}, \{f_1\}, \{f_2, f_3\}, \{f_2\}, \{f_1, f_3\}\}.
\]
In the preceding example, we did not consider any application of a set of equations, \( E \), to a set of trees \( \Theta \). We will now discuss the application of \( E \) over components of \( P(\Theta) \). The idea is to find distinct normal forms of elements of \( \Theta \) under \( E \). These concepts are exemplified below.

3.6.2. Example: introducing \( L(\Theta) \)

Example 3.25

Consider Example 3.14. Let \( T=\{t_1, t_2, t_3, t_4\} \) such that:

\[
\begin{align*}
t_1 &= \emptyset \\
t_2 &= F < aF < c \emptyset > \\
t_3 &= F < aF < bF < aF < \emptyset >> \\
t_4 &= F < bF < aF < a \emptyset >>
\end{align*}
\]

Without considering any equation (i.e., applications of \( E \) to \( T \)), \( P(T) \) has to be pictured as in Figure 3.6. Applying \( E=\{e_1, e_2\} \) of Example 3.14 to elements of \( T \) yield:

\[
\begin{align*}
\llbracket t_1 \rrbracket &= \{t_1\} \\
\llbracket t_2 \rrbracket &= \{t_2, t_2'\} \\
\llbracket t_3 \rrbracket &= \{t_3, t_4\} \\
\llbracket t_4 \rrbracket &= \{t_4', t_3\}
\end{align*}
\]
where

\[ t_2^b = F < c \ F < a \emptyset > > \]
\[ t_3^b = F < a \ F < b \emptyset > > \]
\[ t_4^b = F < b \ F < a \emptyset > > \]

As we see, there are only three distinct representation forms for T. Thus, after applying E to P(T) the resulting components must put \( t_3 \) and \( t_4 \) together in braces (the same class). That is, since \( t_3 \) and \( t_4 \) have the same reduced forms, their meaning must be equal.

Now if we just keep those components of P(T) such that \( t_3 \) and \( t_4 \) are together, then the result will be Figure 3.7 and we will denote it by L(T).

Since \( \emptyset \) is defined both for terminated and nonterminated trees, thus, we can talk about L(\emptyset). We should note that the bottommost element of L(\emptyset) has a property such that the elements in braces ('{', '}') are equivalent and those in different partitions are unequivalent. Intuitively, L(\emptyset) indicates the relationships between \emptyset and E (equations). That is, it shows all of the interpretations that are possible for elements of \emptyset under E. Obviously, L(\emptyset) is a subset of P(\emptyset).

The following definition will be used for construction of L(\emptyset) under E directly. Then, we will prove that L(\emptyset) is a "complete lattice". That L(\emptyset) is a complete lattice indicates that we will not have endless reductions for a tree t in T_\emptyset under E. Furthermore, the bottommost element of L(\emptyset) gives the final representation for \emptyset. That is, equivalent trees fall into one class.
Figure 3.5 $P\{f_1, f_2, f_3\}$ under binary relation $\geq$
\[ \begin{align*}
A &= t_1 \\
B &= t_2 \\
C &= t_3 \\
D &= t_4 \\
\end{align*} \]

Figure 3.6 The lattice with four elements (\(P(T)\))
Figure 3.7 Effective lattice: L(T)
**Definition 3.26** \(L(\emptyset)\)

1. \(\emptyset \in L(\emptyset)\).
2. if \(\pi \in L(\emptyset)\), \(\pi = \{A_1, \ldots, A_j, \ldots, A_k\}\), \(A_j = A_j^1 \cup A_j^2\), \(A_j^1 \cap A_j^2 = \emptyset\), and \([A_j^1] \cap [A_j^2] = \emptyset\), then \(\pi' = \{A_1, \ldots, A_j^1, A_j^2, \ldots, A_k\} \in L(\emptyset)\).
3. nothing else is in \(L(\emptyset)\) unless constructed by (1) and (2) above.

In the following example, we show how to compute the greatest lower bound of \(\pi_1\) and \(\pi_2\) (GLB(\(\pi_1, \pi_2\))) for \(\pi_1, \pi_2\) in \(L(\emptyset)\). Then, we provide a formal definition of GLB(\(\pi_1, \pi_2\)) and LUB(\(\pi_1, \pi_2\)).

**Example 3.27**

Let \(\Theta = \{f_1, f_2, f_3, f_4, f_5, f_6\}\) such that \(f_i \in \Theta\) for \(i = 1, \ldots, 6\). Let \(\pi_1, \pi_2 \in L(\emptyset)\) such that

\[
\pi_1 = \{\{f_1, f_2\}, \{f_3, f_4\}, \{f_5, f_6\}\},
\]

\[
\pi_2 = \{\{f_1, f_4\}, \{f_2, f_3\}, \{f_5\}, \{f_6\}\}.
\]

Then

\[
\text{LUB}(\pi_1, \pi_2) = \{\{f_1, f_2, f_3, f_4\}, \{f_5, f_6\}\},
\]

\[
\text{GLB}(\pi_1, \pi_2) = \{\{f_1\}, \{f_2\}, \{f_3\}, \{f_4\}, \{f_5\}, \{f_6\}\}.
\]

That is, GLB(\(\pi_1, \pi_2\)) is the intersection of every element of \(\pi_1\) with every element of \(\pi_2\), leaving out empty sets, and LUB(\(\pi_1, \pi_2\)) is the union of every element of \(\pi_1\) with every element of \(\pi_2\) if those two elements have a common element. It can be verified that if \(\pi_1\) and \(\pi_2\) are elements of in a set of effective partitions of \(L(\emptyset)\) then GLB(\(\pi_1, \pi_2\)) and LUB(\(\pi_1, \pi_2\)) are unique elements. This property comes from the definition of \(L(T)\) and the binary relation \(\geq\).
Definition 3.28 \([\oplus, \odot]\)

Let \(\pi_1, \pi_2 \in P(\emptyset)\) such that \(\pi_1 = \{A_1, \ldots, A_m\}\) and \(\pi_2 = \{B_1, \ldots, B_n\}\). Then

\[\pi_1 \odot \pi_2 = (Z_1, Z_2, \ldots, Z_k) \in P(\emptyset)\] such that for each \(1 \leq k \leq k\), \(Z_k\) is the smallest set such that
1. \(Z_k \cap A_j = A_j\) or \(\emptyset\) for \(1 \leq j \leq m\); and
2. \(Z_k \cap B_j = B_j\) or \(\emptyset\) for \(i \leq j \leq n\).

\[\pi_1 \oplus \pi_2 = (Z_1, Z_2, \ldots, Z_k) \in P(\emptyset)\] such that for each \(1 \leq k \leq k\), \(Z_k\) is the largest set such that
1. \(Z_k \cap A_j = Z_k\) or \(\emptyset\) for \(1 \leq j \leq m\); and
2. \(Z_k \cap B_j = Z_k\) or \(\emptyset\) for \(1 \leq j \leq n\).

The preceding defines the operations \(\oplus\) and \(\odot\). We now state and prove some properties of \(\oplus\) and \(\odot\).

Lemma 3.29

For all \(\pi_1, \pi_2\) in \(P(\emptyset)\),

(a) \(\pi_1 \odot \pi_2 \geq \pi_1, \quad \pi_1 \oplus \pi_2 \geq \pi_2\)

(b) \(\pi_1 \odot \pi_2 \geq \pi_2 \odot \pi_1\)

(c) \(\pi_1 \preceq \pi_2 \odot \pi_1, \quad \pi_1 \odot \pi_2 \preceq \pi_1 \odot \pi_2\)

(d) \(\pi_1 \odot \pi_2 = \pi_2 \odot \pi_1\)

**Proof:** Suppose \(\pi_1 = \{A_1, \ldots, A_m\}\), \(\pi_2 = \{B_1, \ldots, B_n\}\), and \(\pi_1 \odot \pi_2 = \{Z_1, \ldots, Z_k\}\).

(a) the proof of \(\pi_1 \odot \pi_2 \geq \pi_1\) and \(\pi_1 \odot \pi_2 \geq \pi_2\) is immediate from Definition 3.28, since each \(A_j\) or \(B_j\) is a subset of the element of \(Z_j\).
(b) for proving part (b), we have $\pi_1 \circ \pi_2 \succeq \pi_2$ and $\pi_1 \circ \pi_2 \succeq \pi_1$. Now, let $\pi_2 \succeq \pi_1$ and $\pi_2 \succeq \pi_2$. Then $\pi_2 \circ \pi_1 \circ \pi_2$. Hence, $\pi_1 \circ \pi_2 \succeq \pi_2 \circ \pi_1$. Also, we have $\pi_1 \circ \pi_2 \succeq \pi_1$ and $\pi_1 \circ \pi_2 \succeq \pi_2$; hence, $\pi_2 \circ \pi_1 \succeq \pi_1 \circ \pi_2$.

Thus, by antisymmetry $\pi_1 \circ \pi_2 = \pi_2 \circ \pi_1$.

(c, d) Proof of parts (c) and (d) are left to the reader. □

We exemplify the Definition 3.28 by the following example.

Example 3.30

Let $\Theta = \{f_i | f_i \in \Sigma$ for $i=1, \ldots, 12\}$, and

\[ \pi_1 = \{\{f_1, f_2, f_3, f_4\}, \{f_5, f_6, f_7\}, \{f_8, f_9\}, \{f_{10}, f_{11}, f_{12}\}\} = \{A_1, A_2, A_3, A_4\} \]

\[ \pi_2 = \{\{f_1, f_2\}, \{f_3, f_4, f_8\}, \{f_9, f_{10}\}, \{f_{11}, f_{12}\}, \{f_5, f_6, f_7\}\} = \{B_1, B_2, B_3, B_4, B_5, B_6\} \]

Then

\[ \pi_1 \circ \pi_2 = \{\{f_1, f_2, f_3, f_4, f_8, f_9, f_{10}, f_{11}, f_{12}\}, \{f_5, f_6, f_7\}\} = \{Z', Z''\} \]

from definition. $\pi_1 \circ \pi_2 = \{Z_1, Z_2, Z_3, Z_4\}$ where $Z_1 = Z_2 = Z' = Z$ and $Z_3 = Z''$

\[
\begin{aligned}
Z' \cap A_1 &= A_1 & i = 1, 3, 4 \\
Z' \cap A_2 &= \emptyset \\
Z' \cap B_1 &= B_1 & i = 1, 2, 3, 4 \\
Z' \cap B_j &= \emptyset & j = 5, 6 \\
Z'' \cap A_1 &= \emptyset & i = 1, 3, 4 \\
Z'' \cap A_2 &= A_2 \\
Z'' \cap B_1 &= \emptyset & i = 1, 2, 3, 4 \\
Z'' \cap B_j &= B_j & j = 5, 6
\end{aligned}
\]
Similarly,
\[ \pi_1 \otimes \pi_2 = \{f_1, f_2, \{f_3, f_4\}, \{f_5\}, \{f_6, f_7\}, \{f_8\}, \{f_9\}, \{f_{10}\}, \{f_{11}, f_{12}\}\}. \]

While Definition 3.28 for \( \otimes \) and \( \ominus \) is simple, it is not constructive. The following two algorithms give a construction for computing \( \pi_1 \otimes \pi_2 \) and \( \pi_1 \ominus \pi_2 \). In order to facilitate the algorithms, we use the following assumptions:

\[ \pi_1 = \{A_1, \ldots, A_m\}, \]
\[ \pi_2 = \{B_1, \ldots, B_n\}, \] and
\[ \pi = \pi_1 \cup \pi_2 \text{ where } \pi = \{x_1, x_2, \ldots, x_{m+n}\}. \]

SUM (Figure 3.8) computes \( \pi_1 \otimes \pi_2 \) and MUL (Figure 3.9) computes \( \pi_1 \ominus \pi_2 \).

The statements in Figure 3.8 and Figure 3.9 enclosed by the symbols \{ and \} are descriptive comments.

**Theorem 3.31**

\[ \text{SUM}(\pi_1, \pi_2) = \pi_1 \otimes \pi_2 \]
\[ \text{MUL}(\pi_1, \pi_2) = \pi_1 \ominus \pi_2. \]

**Proof:** To prove the theorem, first we show that LUB(\( \pi_1, \pi_2 \)) and GLB(\( \pi_1, \pi_2 \)) exist and are in P(\( \emptyset \)).

**Theorem 3.32**

If \( \pi_1, \pi_2 \in P(\emptyset) \), then LUB(\( \pi_1, \pi_2 \)) and GLB(\( \pi_1, \pi_2 \)) exist and are in P(\( \emptyset \)).

**Proof:** To prove the theorem, first we show that LUB(\( \pi_1, \pi_2 \)) exists and is in P(\( \emptyset \)). Before proving the theorem, we restate the definition of \( \pi_1 \otimes \pi_2 \). Let \( \pi_1 = \{A_1, \ldots, A_m\} \) and \( \pi_2 = \{B_1, \ldots, B_n\} \), then \( \pi_1 \otimes \pi_2 = \{Z_1, \ldots, Z_k\} \in P(\emptyset) \) such that for each \( 1 \leq j \leq k \), \( Z_j \) is the smallest set such that
function \text{SUM}(\pi_1, \pi_2: P(\emptyset)): P(\emptyset);

\{The function \text{SUM} computes \pi_1 \Phi \pi_2\}
\{such that \pi_1 = \{A_1, \ldots, A_m\} and \pi_2 = \{B_1, \ldots, B_n\}\}
\{\pi = \pi_1 \cup \pi_2 = \{X_1, \ldots, X_{m+n}\}\}

\begin{align*}
\text{begin} & \quad \text{SUM} := \emptyset; \\
& \quad \text{for } i := 1 \text{ to } m \text{ do} \\
& \quad \quad \text{begin} \\
& \quad \quad \quad \text{Z} := A_i; \\
& \quad \quad \quad \text{for } j := 1 \text{ to } m+n \text{ do} \\
& \quad \quad \quad \quad \text{begin} \\
& \quad \quad \quad \quad \quad \text{if } (Z \cap X_j \neq \emptyset) \\
& \quad \quad \quad \quad \quad \quad \text{then } Z := Z \cup X_j \\
& \quad \quad \quad \quad \quad \text{end;} \\
& \quad \quad \quad \text{SUM} := \text{SUM} \cup \{Z\}; \\
& \quad \quad \text{end;} \\
& \text{end;} \quad \{\text{end of SUM}\}
\end{align*}

Figure 3.8 Algorithm for computing $\pi_1 \Phi \pi_2$
function \text{MUL}(\pi_1, \pi_2; P(\emptyset)) : P(\emptyset);

\{The function \text{MUL} computes \pi_1 \ast \pi_2\}
\{such that \pi_1 = \{A_1, \ldots, A_m\} and \pi_2 = \{B_1, \ldots, B_n\}\}
\{\pi = \pi_1 \cup \pi_2 = \{X_1, \ldots, X_{m+n}\}\}

\text{begin} \text{MUL} := \emptyset;
\text{for} \ i := 1 \text{ to } m \text{ do}
\text{begin}
\quad \text{Z} := A_i
\text{for} \ j := 1 \text{ to } m+n \text{ do}
\quad \text{begin}
\quad \quad \text{if} (Z \cap X_j \neq \emptyset)
\quad \quad \text{then} \text{Z} := Z \cap X_j
\quad \text{end;}
\quad \text{MUL} := \text{MUL} \cup \{Z\};
\text{end;}
\text{end}; \{\text{end of MUL}\}

Figure 3.9 Algorithm for computing \pi_1 \ast \pi_2
1) \( Z_j \cap A_k = A_k \) or \( \emptyset \) for \( 1 \leq k \leq m \); and

2) \( Z_j \cap B_k = B_k \) or \( \emptyset \) for \( 1 \leq k \leq n \).

Now, suppose \( \pi_3 \geq \pi_1 \) and \( \pi_3 \geq \pi_2 \) where \( \pi_3 = \{ C_1, \ldots, C_p \} \in P(\emptyset) \). We claim that \( \pi_3 \geq \pi_1 \phi \pi_2 \). Suppose, by way of contradiction, \( \pi_3 \not\geq \pi_1 \phi \pi_2 \). Then consider the set \( \pi' = \{ D : D = C_1 \cap Z_j, C \in \pi_3 \text{ and } Z_j \in \pi_1 \phi \pi_2 \} \). First, \( \pi' \) is a partition. Secondly, \( \pi_1 \phi \pi_2 \supset \pi' \). Thus, every \( D \) is a subset of some \( Z_j \). Now, consider \( D \cap A_k \) for some \( A_k \), then

\[
D \cap A_k = C_1 \cap Z_j \cap A_k = C_1 \cap A_k \text{ or } C_1 \cap \emptyset
\]

since \( D = C_1 \cap Z_j \)

\[
= C_1 \cap A_k \text{ or } C_1 \cap \emptyset
\]

since \( Z_j \cap A_k = A_k \) or \( \emptyset \) by definition of \( \pi_1 \phi \pi_2 \)

\[
= A_k \text{ or } \emptyset \text{ since } C_1 \cap A_k = A_k \text{ or } \emptyset \text{ because } \pi_3 \geq \pi_1.
\]

Similarly, \( D \cap B_k = B_k \) or \( \emptyset \), and each \( D \) is a subset of some \( Z_j \). Thus, we have found a \( \pi' \in P(\emptyset) \) such that each \( D \) in \( \pi' \) is a subset of some \( Z_j \) in \( \pi_1 \phi \pi_2 \). Thus, \( Z_j \) is not the smallest set with properties (1) and (2) of definition of \( \pi_1 \phi \pi_2 \). This is a contradiction. Hence, \( \pi_3 \geq \pi_1 \phi \pi_2 \). Also, from Lemma 3.29 we have \( \pi_1 \phi \pi_2 \geq \pi_1 \) and \( \pi_1 \phi \pi_2 \geq \pi_2 \).

So far we have:

\[
\pi_1 \phi \pi_2 \geq \pi_1, \pi_1 \phi \pi_2 \geq \pi_2
\]

\[
\pi_3 \geq \pi_1, \pi_3 \geq \pi_2 \text{ imply } \pi_3 \geq \pi_1 \phi \pi_2
\]

Thus, by Definition 3.21, \( \pi_1 \phi \pi_2 = \text{LUB}(\pi_1, \pi_2) \). Next, we prove that \( \pi_1 \phi \pi_2 = \text{GLB}(\pi_1, \pi_2) \). Again, we restate the definition of \( \pi_1 \phi \pi_2 \):

\[
\pi_1 \phi \pi_2 = \{ Z_1, \ldots, Z_r \} \in P(\emptyset) \text{ such that for each } 1 \leq j \leq r, \text{ Z}_j \text{ is the largest set such that:}
\]
1') \( Z_j \cap A_x^i = Z_j \) or \( \emptyset \) for \( 1 \leq i \leq m \); and
2') \( Z_j \cap B_x^i = Z_j \) or \( \emptyset \) for \( 1 \leq i \leq n \).

Now, suppose \( \pi_1 \geq \pi_3, \pi_2 \geq \pi_3 \) where \( \pi_3 = (R_1, \ldots, R_k) \in \mathcal{P}(\Theta) \). We claim that \( \pi_1 \otimes \pi_2 \geq \pi_3 \). Suppose, by way of contradiction, \( \pi_1 \otimes \pi_2 \geq \pi_3 \). Consider the set

\[ \pi'' = \{ D / D = R_i \cup Z_j, \; R_i \in \pi_3, \; Z_j \in \pi_2^{\otimes}, \; \text{and} \; R_i \cap Z_j \neq \emptyset \} \]

Note that \( \pi'' \) is a partition of \( \Theta \) (i.e., \( \pi'' \in \mathcal{P}(\Theta) \)). Now consider \( D \cap A_x^i \) for some \( A_x^i \). Then

\[ D \cap A_x^i = (R_i \cup Z_j) \cap A_x^i \]

since \( D = R_i \cup Z_j \) for some \( Z_j \) with \( R_i \cap Z_j \neq \emptyset \)

\[ = (R_i \cap A_x^i) \cup (Z_j \cap A_x^i) \]

from distribution law in set theory

Now we identify four cases:

**Case 1:** \( R_i \cap A_x^i = R_i \) and \( Z_j \cap A_x^i = Z_j \) so, \( D \cap A_x^i = R_i \cup Z_j = D \)

**Case 2:** \( R_i \cap A_x^i = R_i \) and \( Z_j \cap A_x^i = \emptyset \). This is impossible because \( R_i \cap Z_j \neq \emptyset \)

**Case 3:** \( R_i \cap A_x^i = \emptyset \) and \( Z_j \cap A_x^i = Z_j \). This is impossible because \( Z_j \subseteq A_x^i \)

**Case 4:** \( R_i \cap A_x^i = \emptyset \) and \( Z_j \cap A_x^i = \emptyset \) so, \( D \cap A_x^i = \emptyset \cup \emptyset = \emptyset \)

Thus, \( D \cap A_x^i = D \) or \( \emptyset \). Thus \( D \cap A_x^i = D \) or \( \emptyset \), and each \( D \) is a superset of \( Z_j \) (\( Z_j \subseteq D \)). Thus, \( Z_j \) is not the largest set within the properties (1') and (2'), and this is a contradiction to our assumptions. Hence, \( \pi_1 \otimes \pi_2 \geq \pi_3 \). Also, from Lemma 3.29, we have \( \pi_1 \geq_1 \pi_2 \otimes 2 \) and \( \pi_2 \geq_2 \pi_2 \otimes 3 \).

In conclusion:

\[ \pi_1 \geq_1 \pi_2 \otimes 2, \quad \pi_2 \geq_2 \pi_1 \otimes 2 \]

\[ \pi_1 \geq_1 \pi_2, \quad \pi_2 \geq_2 \pi_3 \]

imply \( \pi_1 \otimes \pi_2 \geq_2 \pi_3 \)
Hence, \( \pi_1 \otimes \pi_2 = \text{GLB}(\pi_1, \pi_2) \).

Note that the proof of Theorem 3.32 gives the corollary below.

**Corollary 3.33**

Let \( \pi_1, \pi_2 \in \mathcal{P}(\emptyset) \). Then \( \text{LUB}(\pi_1, \pi_2) = \pi_1 \vee \pi_2 \) and \( \text{GLB}(\pi_1, \pi_2) = \pi_1 \otimes \pi_2 \).

**Theorem 3.34**

\( \mathcal{P}(\emptyset) \) is a lattice.

**Proof:** By Theorem 3.32, since \( \text{GLB}(\pi_1, \pi_2) \) and \( \text{LUB}(\pi_1, \pi_2) \) exist for all \( \pi_1, \pi_2 \in \mathcal{P}(\emptyset) \) then by Definition 3.22, \( \mathcal{P}(\emptyset) \) is a lattice.

3.6.3. Basic results about \( L(\emptyset) \)

Figure 3.7 indicates that \( L(\emptyset) \) is exactly a subset of \( \mathcal{P}(\emptyset) \), and elements of \( L(\emptyset) \) have the same relationships to each other as they have in \( \mathcal{P}(\emptyset) \). Also, Theorem 3.32 indicates that \( \mathcal{P}(\emptyset) \) is a lattice since \( \text{GLB}(\pi_1, \pi_2) \) and \( \text{LUB}(\pi_1, \pi_2) \) exist for all \( \pi_1, \pi_2 \) in \( \mathcal{P}(\emptyset) \). Furthermore, it will be proved that if \( \mathcal{P}(\emptyset) \) is a lattice then the subset of \( \mathcal{P}(\emptyset) \) (i.e., \( L(\emptyset) \)) is a lattice. Before proving that \( L(\emptyset) \) is a lattice, we define a "sublattice".

**Definition 3.35 [Sublattice]**

Let \( (L, \leq) \) be a lattice and \( \mathfrak{A} \subseteq L \). Then \( (\mathfrak{A}, \leq) \) is a "sublattice" of \( L \) if and only if \( \mathfrak{A} \) is closed under both operations \( \text{GLB} \) and \( \text{LUB} \).

Definition 3.35 indicates that a sublattice itself is a lattice.

**Theorem 3.36**

If \( \pi_1, \pi_2 \in L(\emptyset) \subseteq \mathcal{P}(\emptyset) \), then \( \text{GLB}(\pi_1, \pi_2) \) and \( \text{LUB}(\pi_1, \pi_2) \) are in \( L(\emptyset) \).
Proof: We have already proved that $\text{LUB}(\pi_1, \pi_2) = \pi_1 \wedge \pi_2$ and $\text{GLB}(\pi_1, \pi_2) = \pi_1 \vee \pi_2$. To prove the theorem we have to show that $\pi_1 \wedge \pi_2$ and $\pi_1 \vee \pi_2$ are in $L(\emptyset)$ for $\pi_1, \pi_2 \in L(\emptyset)$. Let $\pi_1 = \{A_1, \ldots, A_m\}$, $\pi_2 = \{B_1, \ldots, B_n\}$.

Suppose $\pi_1 \wedge \pi_2 = \{Z_1, \ldots, Z_p\}$ is not in $L(\emptyset)$. Thus, there exist $Z_i, Z_j$ such that $Z_i \cap Z_j = \emptyset$ and $[Z_i] \cap [Z_j] \neq \emptyset$. On the other hand, we know that $\pi_1 \geq \pi_1$ and $\pi_2 \geq \pi_2$. Let $A_{i_1}, A_{i_2}, \ldots, A_{i_k} \in Z_i$ such that there is no $A_{k+1} \in Z_i$ where $Z_i = A_{i_1} \cup A_{i_2} \cup \ldots \cup A_{i_k}$. Also, let $A_{j_1}, A_{j_2}, \ldots, A_{j_r} \in Z_j$ such that there is no $A_{j_{r+1}} \in Z_j$ where $Z_j = A_{j_1} \cup A_{j_2} \cup \ldots \cup A_{j_r}$. Also, we know that $[[A_{i_1}]] \cap [A_{j_1}] = \emptyset$ for all $i, j \in \{1, \ldots, m\}$. Since $[Z_i] \cap [Z_j] \neq \emptyset$, then there must exist $A_i' \in Z_i$ and $A_j' \in Z_j$ such that $[[A_i']] \cap [A_j'] \neq \emptyset$. This implies that $\pi_1 \not\subseteq L(\emptyset)$, and this is a contradiction. Hence, $\pi_1 \wedge \pi_2$ must be in $L(\emptyset)$. Proving that $\pi_1 \vee \pi_2$ is in $L(\emptyset)$ is similar to proving that $\pi_1 \wedge \pi_2$ is in $L(\emptyset)$ and will not be repeated here.

In Theorem 3.36, we proved that $L(\emptyset)$ is a lattice. We now extend the definition of GLB and LUB to a subset of poset. Later, we will use the following definition to prove that $L(\emptyset)$ (or $L(T)$) is a complete lattice.

**Definition 3.37 [GLB(A), LUB(A)]**

If $(L, \succeq)$ is any poset and $A$ is any subset of $L$, then element $c$ in $L$ is a least upper bound (LUB) of $A$ if:

(i) $c \succeq a$ for all $a \in A$;

(ii) $x \succeq a$ for all $a \in A$ implies $x \succeq c$; and $c$ is designated by $c = \text{LUB}(A)$.

Dually, $d$ in $L$ is a greatest lower bound (GLB) of $A$ if:
(i)' \( a \geq d \) for all \( a \in A \);

(ii)' \( a \geq x \) for all \( a \in A \) implies \( d \geq x \); and \( d \) is designated by

\[ d = \text{GLB}(A). \]

Lemma 3.38

If \( A = \{ \pi_1, \ldots, \pi_n \} \) and \( n > 1 \) then \( \text{LUB}(A) = \text{LUB}(\ldots \text{LUB}(\text{LUB}(\pi_1, \pi_2), \pi_3) \ldots, \pi_n) \).

Proof: The proof is by induction on \( |A| \). If \( |A| = 1 \) then \( A = \{ \pi_1 \} \). Then \( \text{GLB}(A) = \text{LUB}(A) = \pi_1 \) follows from the reflexivity of \( \geq \) and the definition of \( \text{GLB} \) and \( \text{LUB} \). Now, let \( |A| = 3 \) and \( A = \{ \pi_1, \pi_2, \pi_3 \} \). To show that \( \text{LUB}(A) \) exists, set \( d = \text{LUB}(\pi_1, \pi_2) \), \( e = \text{LUB}(d, \pi_3) \). We claim that \( e = \text{LUB}(A) \). Since \( d \geq \pi_1 \), \( d \geq \pi_2 \), and \( e \geq \pi_3 \); therefore, by transitivity \( e \geq a \), for all \( a \in A \). Also, assume \( b \geq \pi_1 \), \( b \geq \pi_2 \) then \( b \geq d \); also \( b \geq \pi_3 \) so that \( b \geq \pi_3 \), \( b \geq d \); therefore, \( b \geq e \), since \( e = \text{LUB}(d, \pi_3) \). Thus, \( e \) is the \( \text{LUB}(A) \). The proof of inductive step is left to the reader. □

Theorem 3.39

If \( x \in \mathbb{L}(\emptyset) \), then \( \text{GLB}(x) \) and \( \text{LUB}(x) \) are in \( \mathbb{L}(\emptyset) \). □

Theorem 3.40

\((\mathbb{L}(\emptyset), \geq)\) is a complete lattice. □

Theorem 3.41

Let \( \pi_b = \{ Z_1, \ldots, Z_i, \ldots, Z_k \} \) be the least element of \( \mathbb{L}(\emptyset) \) (i.e., \( \pi_b \geq \pi \) for all \( \pi \in \mathbb{L}(\emptyset) \)). Then \( \{ [Z_i] \} \cap \{ [Z_j] \} = \emptyset \) for \( i \neq j \) and for each \( f \in Z_i \), \( [f] = [Z_i] \).
Proof: The property of \( \| Z_i \| \cap \| Z_j \| = \emptyset \) for \( i \neq j \) is immediate from the definition of \( L(\emptyset) \). To prove \( \| f \| = \| Z_i \| \) for \( f \in Z_i \), suppose \( \| f \| \neq \| Z_i \| \); then by definition of \( L(\emptyset) \), there must exist
\[ \pi' = (Z_1, \ldots, Z_{i-1}, Z'_i, Z_i', Z_{i+1}, \ldots, Z_k) \in L(\emptyset) \]
such that \( Z_i = Z'_i \preceq Z''_i \) where \( \| Z'_i \| = \| f \| \) and \( \| Z'_i \| \cap \| Z''_i \| = \emptyset \). Thus \( \pi_b \succeq \pi' \), since every element in \( \pi' \) is a complete subset of some element in \( \pi_b \). This contradicts the assumption of the statement of this theorem, which states that \( \pi \succeq \pi_b \) for all \( \pi \in L(\emptyset) \).

\[ \Box \]

3.6.4. Algorithm for construction of \( L(\emptyset) \)

We now give an algorithm for construction of \( L(\emptyset) \). Suppose, we are given a finite set \( \emptyset \) of nonterminated trees (or terminated trees), and a routine for computing the reduced forms of each member of \( \emptyset \). That is, given a \( f \in \emptyset \), we assume that \( \| f \| \) can be found under equations E. Then our task is to find \( L(\emptyset) \). First, we will give the general algorithm (possibly nonterminating) for construction of \( L(\emptyset) \). Then, we will impose some sufficient conditions on tree rewriting system (E), which will guarantee the termination of algorithm.

The technique for finding \( L(\emptyset) \) is to repeatedly refine the blocks (elements) of the original partition \( \{\emptyset\} \) by the following method. Let \( B \) be a block. Examine \( \| b \| \) for each \( b \) in \( B \). Then partition \( B \) in such a way that \( B = B' \cup B'' \) where \( B' \cap B'' = \emptyset \). The process is iterated until no further refinement are possible.
Let $\Theta=\{f_1, \ldots, f_n\}$, $n>0$, $f_i \in \Sigma$, for $i=1, \ldots, n$; and assume the existence of a (possibly noncomputable) function TEST such that

$$\text{TEST}(f_i, f_j) = \begin{cases} \text{true} & \text{if } \llbracket f_i \rrbracket = \llbracket f_j \rrbracket \\ \text{false} & \text{otherwise} \end{cases}$$

We will use the "unaryandbinaryrefinement" function (Figure 3.10) for construction of $L(\Theta)$. The unaryandbinaryrefinement operates on any nonempty subset of $\Theta$ and finds all of the partitions which satisfies the requirements of $L(\Theta)$. We will use the convention that any text enclosed by braces { and } is a comment.

We next define a function "allrefinements", which successively finds the elements of $L(\Theta)$ by using the unaryandbinaryrefinement's function. The function "allrefinements" is shown in Figure 3.11.

**Lemma 3.42**

Unaryandbinaryrefinement ($X$) is either $\{X\}$ or $\{n/n\}$ is a binary partition $\{X_1, X_2\}$ of $X$ such that $\llbracket X_1 \rrbracket \cap \llbracket X_2 \rrbracket = \emptyset$.

**Proof:** Let $X=\{x_1, \ldots, x_k\}$ and $x$ be a fixed element of $X$. After applying the "unaryandbinaryrefinement" to $X$ we have:

$$\text{EQ} = \{x_i / \text{TEST}(x_i, x) = \text{true}, x_i \in X\};$$

$$\text{NONEQ} = X - \text{EQ}.$$  

From the above identities, we can see that $\text{EQ} \cap \text{NONEQ} = \emptyset$. Now, if $\text{NONEQ} = \emptyset$ or $\text{EQ} = \emptyset$ then unaryandbinaryrefinement($X$) = $\{X\}$; otherwise $\text{NONEQ} \neq \emptyset$ and $\text{EQ} \neq \emptyset$. We set $X_1 = \text{EQ}$ and $X_2 = \text{NONEQ}$. Furthermore, definition of the TEST function guarantees that $\llbracket X_1 \rrbracket \cap \llbracket X_2 \rrbracket = \emptyset$. \qed
function unaryandbinaryrefinement(X: set): partitions;
{X is a nonempty subset of \( \emptyset = \{f_1, \ldots, f_n\} \) }
{unaryandbinaryrefinement(X) is the set of binary }
{partitions of X with disjoint reduced forms }
{or just X itself if X has no such binary partitions}
begin
refine:=\emptyset \quad \text{(refine is a partition)}
for each element \( f_i \) in X do
begin \{EQ and NONEQ are binary partitions of X \}
EQ:=\emptyset;
NONEQ:=\emptyset;
for each element \( f_j \) in X do
begin
if \( \text{TEST}(f_i, f_j) \)
then EQ:=EQ u\{f_j\}
else NONEQ:=NONEQ u\{f_j\};
end;
refine:=refine u\{(EQ,NONEQ)-\emptyset\};
end;
unaryandbinaryrefinement:=refine;
end; \{end of unaryandbinaryrefinement function\}

Figure 3.10 Unaryandbinaryrefinement function
function allrefinements (\( \emptyset = \text{set} \)): poset of partitions;

\{the allrefinement function finds the elements of \( L(\emptyset) \)\}

begin
1. if (\(|\emptyset| = 1\))
2. then allrefinements:=\{\emptyset\}
3. else begin {put both \( \emptyset \) and its refinements into PL}
4. PL:=\{\{\emptyset\}\} u unaryandbinaryrefinement (\( \emptyset \))
5. repeat
6. allrefinements:=PL;
   {for each element of "allrefinements",
   {add all possible refinements
   {to "allrefinements".
   }
7. for each \( A = \{A_1, ..., A_r\}\) \( \in \) allrefinements do
8.     PL:=PL u\{a_1 u a_2 u ... a_r / a_i \in \text{allrefinements} \( (A_i) \)
      for \( i=1, ..., r \)\}
9. until (allrefinements = PL);
end;
end; {end of the allrefinements function}

Figure 3.11 Allrefinements function
Theorem 3.43

The algorithm allrefinements (\( \emptyset \)) computes \( L(\emptyset) \).

Proof: The key of the "allrefinements" algorithm is the function "unaryandbinaryrefinement" with the property described in Lemma 3.42.

The proof proceeds by considering the following cases.

Case 1: \(|\emptyset|=1\), then the algorithm allrefinements computes \( L(\emptyset) \) and terminates, and the conditions of \( L(\emptyset) \) are met.

Case 2: \(|\emptyset|=n, n>1\), and \( \emptyset=\{f_1, \ldots, f_n\} \). In the first pass to the algorithm, after execution of the lines 4-6 (Figure 3.11), we will have:

\[
PL=\{\emptyset, \{EQ_1, \text{NONEQ}_1\}, \ldots, \{EQ_n, \text{NONEQ}_n\}\}
\]

where \( EQ_i \cup \text{NONEQ}_i = \emptyset \), \( EQ_i \cap \text{NONEQ}_i = \emptyset \), and \( \# EQ_i \cap \# \text{NONEQ}_i = 0 \) for \( i=1, \ldots, n \) from Lemma 3.42.

We note that, the unaryandbinaryrefinement function computes all of the binary partitions, which satisfy the requirements of the definition of \( L(\emptyset) \). The lines 7-8 (Figure 3.11) examines each element of "allrefinements" and checks whether each element of the "allrefinements" can be refined furthermore; and if so, it finds all of the partitions which satisfy \( L(\emptyset) \), \( L(EQ_i) \), and \( L(\text{NONEQ}_i) \) requirements.

Line 9 (Figure 3.11) checks whether lines 7-8 have added additional elements to "allrefinement", if so then it repeats the process from line 5 and if not then it means that no further refinements are possible, and we have to stop. Line 6 of Figure 3.11 guarantees that each time in the iteration we are refining the whole possible
elements for generating the remaining elements of L(θ). Since for each set we find the maximum possible binary and unary partitions; thus, it is impossible to exclude any element from L(θ). □

Example 3.44 illustrates the construction of L(θ) by executing all-refinements(θ).

Example 3.44

Let θ={f_1, f_2, f_3, f_4} and [[f_3]]=[[f_4]]. Applying algorithm all-refinements to θ will result:

\[ PL = \{\emptyset\} \cup \text{unaryandbinaryrefinement}(\emptyset) \]
\[ = \{\emptyset, \{f_1\}, \{f_2, f_3, f_4\}, \{f_2\}, \{f_1, f_3, f_4\}, \{f_1, f_2\}, \{f_3, f_4\}\} \]

Now, executing lines 7-8 (Figure 3.11) for each member of L say \( A_1=\{f_1\}, \{f_2, f_3, f_4\} \) will result:

\[ L(A_1) = \{f_1\} \text{ since } A_1=\{f_1\}; \text{ and} \]
\[ L(A_2) = \{f_2, f_3, f_4\} \text{ since } A_2=\{f_2, f_3, f_4\}. \]

Thus \( a_1 \cup a_2 = \{f_1\}, \{f_2\}, \{f_3, f_4\} \).

Similarly, repeating the above process for other members of L, we will have:

\[ PL = \{\emptyset, \{f_1\}, \{f_2, f_3, f_4\}, \{f_2\}, \{f_1, f_3, f_4\}, \{f_1, f_2\}, \{f_3, f_4\}\} \]

and allrefinements=PL.

Now, the refinements of elements of L will not add new elements to PL. Hence, L=PL and this terminates the computation. Thus, "allrefinements" correctly computes L(θ). □
3.6.5. Conditions for compatibility of \( L(\Theta) \)

So far in construction of \( L(\Theta) \), we have assumed the existence and decidability of the function \( \text{TEST} \). In general, it has been shown that the \( \text{TEST} \) function is uncomputable for tree rewriting systems \([16, 27]\). Thus, we have to consider a class of rewriting systems such that \( \text{TEST} \) function is computable. Hence, a basic question which must be answered in the construction of \( L(\Theta) \) is what conditions must be imposed on \( \Theta \) and the tree rewriting system to guarantee the decidability of the \( \text{TEST} \) function and hence the halting of algorithm "allrefinements" for computation of \( L(\Theta) \).

In \([16]\), they have imposed sufficient syntax-checkable conditions on tree rewriting systems, which guarantees the termination of tree rewriting system, and hence the \( \text{TEST} \) function becomes computable. Briefly, we will address the problem of showing under what conditions the \( \text{TEST} \) function is computable. In order to talk about those sufficient conditions and deal with the \( \text{TEST} \) function in terms of \( \Rightarrow^* \), we need to introduce the following concepts.

Let \( E \) be a tree rewriting system associated with \( \Sigma \), then a "nonterminating" reduction sequence from \( f \) is an infinite sequence \( f \Rightarrow f_1 \Rightarrow \ldots \). The reduction relation (i.e., \( \Rightarrow_E \)) is said to be "terminating" for \( f \) if there is no nonterminating reduction sequence from \( f \). Let \( \Theta \subseteq E \) be a finite set, then "termination over \( \Theta \) is decidable" if \( \Rightarrow \) is terminating for every \( f \) in \( \Theta \).
Definition 3.45  [Globally Finite Relation]

A reduction relation $\Rightarrow$ is "globally finite" if for every $f$ in $\Theta \subseteq \Xi$ the set of $f'$ such that $f \stackrel{*}{\Rightarrow} f'$ is finite. □

Shortly, we will see that global finiteness of $\Rightarrow$ and finiteness of $\Theta$ are sufficient conditions for computability of the TEST function. In [16, 22] they have proved the following lemmas.

Lemma 3.46

If a rewriting relation $\Rightarrow$ is globally finite and $\Theta$ is finite, termination over $\Theta$ is decidable.

Proof: for each element of $\Theta$, we just have to follow all possible rewriting sequences until we either reach a terminal form (normal form) or a cycle. By global finiteness, there can be only finitely many such sequences and each is of finite length. □

Lemma 3.47

There is no decision procedure for global finiteness of tree rewriting systems. □

Lemma 3.46 implies that if $\Rightarrow$ is globally finite and $\Theta$ is finite then termination over $\Theta$ is decidable. Termination over $\Theta$ indicates that we are able to compute $[f]$ for every $f$ in $\Theta$. Hence, if the conditions of Lemma 3.46 are satisfied then the TEST function becomes computable. So, the only thing we need to find, are sufficient conditions for global finiteness of $\Rightarrow$. We need the following concepts from [16] for deriving those sufficient conditions.
Definition 3.48 [Size of Tree]

The size of a tree \( f \in \mathcal{Z} \) is the number of function (operation) and variable symbols it contains. Denoting this \( \text{SIZE}(f) \). For example, we have \( \text{SIZE}(f[g(x_1 x_2 x_3)]) = 5 \).

Definition 3.49 [Nonexpanding Rewrite Rule]

A rewrite rule \( L = R \) is "nonexpanding" if \( \text{SIZE}(L) \geq \text{SIZE}(R) \). Furthermore, a tree rewriting system \( (E) \) is nonexpanding if and only if every rewrite rule in \( E \) is nonexpanding.

We can now state the following Lemma proved in [16].

Lemma 3.50

If a rewriting relation \( \Rightarrow \) is nonexpanding, it is globally finite.

From Lemma 3.50, we can derive the conditions for computability of \( L(\Theta) \) using the algorithm allrefinements (\( \Theta \)) in Lemma 3.51 and Theorem 3.52 below.

Lemma 3.51

Let the rewriting relation \( \Rightarrow \) be globally finite and \( \Theta = \{f_1, \ldots, f_n\} \), \( n > 0 \), be a finite set. Then the TEST function (equality of reduced forms) is computable.

Proof: Lemma 3.46 implies that the termination over \( \Theta \) is decidable. This implies that \( [f_i] \) is computable for \( i = 1, \ldots, n \). Furthermore, global-finiteness of \( \Rightarrow \) indicates that \( [f_i] \) is a finite set.
Thus, given any $f_i$ and $f_j$, it is possible to write an algorithm which can test whether $\llbracket f_i \rrbracket = \llbracket f_j \rrbracket$ because $\llbracket f_i \rrbracket$ and $\llbracket f_j \rrbracket$ are finite. Hence, the function TEST is computable.

**Theorem 3.52**

Algorithm “allrefinements” halts if $\emptyset$ is finite and the reduction relation $\Rightarrow$ is globally-finite.

**Proof:** Let $\emptyset$ be a finite set and $\Rightarrow$ be a globally-finite. Then from Lemma 3.51, the TEST function (function for testing equality of reduced forms) is computable. Hence PL of Figure 3.11 is computable. Thus, the algorithm “allrefinements” halts if $\emptyset$ is finite and the reduction relation $\Rightarrow$ is globally-finite.

3.7. Conclusions

We have introduced rewrite rules as directed equations. It provides a formal way of simplifying the elements of data structures to obtain "reduced forms". An equation ($L=R$) is considered as a rewrite rule ($R$ is usually simpler than $L$) and one simplifies any tree having the form of $L$ to the form $R$. Thus, for $\Xi$ the set of rewriting rules can form a systematic method for simplifying elements (for example, trees) of data structures and rewriting rules enable us to obtain a lattice for a set of trees in $\Xi$.

We proved that under a binary relation ($\triangleright$) the set of all nonempty partitions of a set of trees $\emptyset \in \Xi$ form a lattice, called $P(\emptyset)$. Then by using the concept of reduced forms we described a sublattice $L(\emptyset)$.
of $P(\emptyset)$. Furthermore, we proved that $L(\emptyset)$ is a complete lattice. We found that the least element of $L(\emptyset)$ is the most useful one, since it is the element where the "equivalent" elements of $\emptyset$ are put in the same block.

We have investigated an algorithm for construction of $L(\emptyset)$, given a function for test of equality of reduced forms (TEST). In general the TEST function is uncomputable. But, we investigated sufficient conditions to be imposed on rewrite rules to guarantee that the function TEST (function of equality of reduced forms) is computable. We should note that imposing those sufficient conditions restricts the class of rewrite rules. In Chapter 5, we will work with more powerful rewriting rules, called conditional rewriting rules (or Horn clauses), which are more generally applicable. Before delving into that, however, we study tree transducers and the relation between tree rewriting systems and tree transducers in Chapter 4.
4. NATURAL TREE TRANSDUCERS

In this chapter, we will describe "tree transducers" in terms of homomorphisms between algebras. Our tree transducers will transform syntax trees of one grammar into syntax trees of another grammar.

4.1. Introduction to Tree Transducers

In this section, we will be concerned with the concept of tree transducers. Basically, a tree transducer is a machine that takes a tree as an input and produces another tree as an output [7]. The concept of a tree transducer is crucial in our developments for implementation of data structures.

4.1.1. Construction of a tree transducer within $L(\Theta_1)$ and $L(\Theta_2)$

This chapter is concerned with the translation (implementation of a data structure by another one) and construction of "natural tree transducers". Here, we focus on translations which are specified by a finite set of rules, a tree transducer. Our first task is to examine how to construct a "natural" tree transducer from Lattices $L(\Theta_1)$ and $L(\Theta_2)$. That is, given $\theta_i \leq \Theta_i$, $L(\Theta_i)$ for $i=1,2$; and their least elements $L_1$ and $L_2$ respectively, does there exist a translation from $T_{\Sigma_1}$ to $T_{\Sigma_2}$? Can we use $L_1$ and $L_2$ to find that translation. That is, can we produce the finite set of rules (tree transducer) defining that translation?
After establishing the context for tree transducers, we will see in Chapter 6 to determine the implementation of a source data structure by a target data structure. To do this, however, we examine issues regarding "consistency" of the constructed tree transducer within the equations (tree rewriting systems) in this chapter.

This chapter presents a method for giving the finite specification of a tree transducer, using the least elements of $L(\Theta_1)$ and $L(\Theta_2)$. To illustrate the construction of a tree transducer (finite set of rules) and in order to exemplify the concept of construction of a tree transducer using the least elements of $L(\Theta_1)$ and $L(\Theta_2)$, we present the following examples.

4.1.2. Informal construction of a tree transducer

Example 4.1

In this example we show how to construct a tree transducer using the least elements of $L(\Theta_1)$ and $L(\Theta_2)$. Consider $N_i, \Sigma_i, \varepsilon_i$ and $E_i$ for $i=1,2$ such that:

$N_1 = \{\text{SET, DATA, BOOLEAN}\} = \{S,D,B\}$, \[ \Sigma_1 = \{\text{signature}\} \]

$N_2 = \{\text{LIST, ATOMS}\} = \{L,A\}$

$\varepsilon_1(\lambda,S) = \{\emptyset\}$

$\varepsilon_1(\lambda,D) = \{0,a,b,c,d,e,\ldots,z\}$

$\varepsilon_1(\text{DS},S) = \{\text{INSERT, REMOVE}\}$

$\varepsilon_1(S,D) = \{\text{SIZE}\}$

$\varepsilon_2(\lambda,L) = \{\text{NIL}\}$

$\varepsilon_2(L,L) = \{\text{REVERSE}\}$

$\varepsilon_2(A,L) = \{\text{MAKE}\}$

$\varepsilon_2(LL,L) = \{\text{APPEND}\}$
\[ \Sigma_1(D,D) = \{\text{Succ}\} \quad \Sigma_2(\lambda,\lambda) = \{\text{Cons}\} \quad \Sigma_2(\lambda, A) = \{0, 1, \ldots, 9\} \]

As before, in specifying the equations of \( E_1 \) and \( E_2 \) we will eliminate the domain of equations (the complete version of set data structure will be given again in Chapter 5).

\textbf{equations } \( E_1 \):

\begin{enumerate}
    \item \( \text{INSERT}[X_1 \text{INSERT}[X_2 X_3]] = \text{INSERT}[X_2 \text{INSERT}[X_1 X_3]] \)
    \item \( \text{INSERT}[X_1 \text{INSERT}[X_1 \emptyset]] = \text{INSERT}[X_1 \emptyset] \)
    \item \( \text{REMOVE}[X_1 \text{INSERT}[X_1 X_2]] = \text{REMOVE}[X_1 X_2] \)
    \item \( \text{REMOVE}[X_1 \text{REMOVE}[X_1 X_2]] = \text{REMOVE}[X_1 X_2] \)
    \item \( \text{SIZE}[\emptyset] = 0 \)
    \item \( \text{SIZE}[\text{INSERT}[X_1 X_2]] = \text{Succ} [\text{SIZE}[X_2]] \)
\end{enumerate}

\textbf{equations } \( E_2 \):

\begin{enumerate}
    \item \( \text{APPEND}[\text{NIL} X_1] = X_1 \)
    \item \( \text{APPEND}[\text{MAKE}[X_1 \text{NIL}]] = \text{MAKE}[X_1] \)
    \item \( \text{APPEND}[\text{APPEND}[\text{MAKE}[X_1 X_2] X_3]] = \text{APPEND}[\text{MAKE}[X_1 \text{APPEND}[X_2 X_3]]] \)
    \item \( \text{CONS}[X_1 X_2] = \text{APPEND}[\text{MAKE}[X_1] X_2] \)
    \item \( \text{REVERSE}[\text{NIL}] = \text{NIL} \)
    \item \( \text{REVERSE}[\text{CONS}[X_1 X_2]] = \text{APPEND}[\text{REVERSE}[X_2] \text{CONS}[X_1 \text{NIL}]] \)
    \item \( \text{REVERSE}[\text{REVERSE}[X_1]] = X_1 \)
\end{enumerate}

Now let \( \Theta_1 = \{f_1, f_2, f_3, f_4, f_5, f_6\} \subseteq \Sigma_1 \) and \( \Theta_2 = \{f'_1, f'_2, f'_3, f'_4, f'_5\} \subseteq \Sigma_2 \) where:

\begin{align*}
    f_1 &= \text{INSERT}[X_1 X_2] \\
    f_2 &= \text{INSERT}[X_1 \text{INSERT}[X_2 \emptyset]]
\end{align*}
\[ f_3 = \text{INSERT}[x_2 \text{INSERT}[x_1 \emptyset]] \]
\[ f_4 = \text{INSERT}[a \text{ INSERT}[b x_3]] \]
\[ f_5 = \text{INSERT}[x_2 \text{INSERT}[x_1 \text{INSERT}[x_3 \emptyset]]] \]
\[ f_6 = \text{INSERT}[b \text{ INSERT}[a x_3]] \]

\[ f'_1 = \text{CONS}[x_1 x_2] \]
\[ f'_2 = \text{CONS}[x_1 \text{APPEND}[\text{NIL} x_2]] \]
\[ f'_3 = \text{CONS}[x_1 \text{CONS}[x_2 \emptyset]] \]
\[ f'_4 = \text{CONS}[2 \text{CONS}[3 x_3]] \]
\[ f'_5 = \text{APPEND}[\text{MAKE}[x_1] \text{MAKE}[x_2]] \]

Let \( L_i \) be the least element of \( L(\emptyset_i) \) for \( i=1,2 \), then
\[ L_1 = \{ (f'_1, f'_2, f'_3, f'_5), \{ f'_4, f'_6 \} \} = \{ A_1, A_2, A_3 \} \]
\[ L_2 = \{ (f'_1, f'_2), \{ f'_3, f'_5 \}, \{ f'_4 \} \} = \{ B_1, B_2, B_3 \} \]

Figure 4.1 shows the pictorial representation of \( L(\emptyset_1) \) and \( L(\emptyset_2) \). Figure 4.1 gives an informal picture for construction of a tree transducer for implementation of a source data structure (set) by target data structure (list). Also, in Figure 4.2 we picture the informal tree transducer by using arrows (\( \rightarrow \)). That is, we associate each element of \( L_1 \) (the least element of \( L(\emptyset_1) \)) to a unique element of \( L_2 \) (the least element of \( L(\emptyset_2) \)) (bijectons).

We can construct many tree transducers, regarding Figure 4.2. Informally, for example \( T_1 \) and \( T_2 \) (below) are two possible tree transducers, which are equivalent.
Figure 4.1 $L(Q_1)$ and $L(Q_2)$
<table>
<thead>
<tr>
<th>Row</th>
<th>Possible Source Trees</th>
<th>Possible Target Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>{f_1}</td>
<td>{f'_1, f'_2}</td>
</tr>
<tr>
<td>(2)</td>
<td>{f_2, f_3, f_5}</td>
<td>{f'_3, f'_5}</td>
</tr>
<tr>
<td>(3)</td>
<td>{f_4, f_6}</td>
<td>{f'_4}</td>
</tr>
</tbody>
</table>

Figure 4.2 Picture of an informal tree transducer
4.1.3. Tree transducers, tree translation

Tree transducers are mechanisms which transform trees into other trees. Tree transducers are used to define the translation of trees from one language into another.

Tree transducers have been examined by several researchers, including Engelfriet [7], Thatcher [40], and Baker [3] to mention a few. Our development follows that of Krishnaswamy and Strawn [31], who investigated conditions to be imposed on tree transducers to guarantee that they induce semantic-preserving translations.

Our algebraic tree transducers process their inputs in a bottom-up (frontier-to-root) fashion. It reads a subtree (source-tree of a row of a tree transducer) of the input tree at a time and produces its output tree (target-tree of the same row of the tree transducer).

Example 4.2

Consider tree transducer $T_2$ of Example 4.1 in this chapter. In this example, we illustrate translation table for an algebraic
tree transducer whose source trees are in $\Sigma_1$ and target trees are in $\Sigma_2$. Figure 4.3 is a translation table for $T_2$. Informally in Figure 4.3, for each $X_i$ we draw an arrow from $X_i$ of source tree to $X_i$ of target tree to show the attachment of the transduction of subtrees at the leaf nodes of source tree to target tree.

The tree transduction performed by a tree transducer $T$ is defined to be a set of ordered pairs of trees $<t_1,t_2>$ such that $T$ can produce output $t_2$ from input $t_1$. Figure 4.4 illustrates a transduction by the tree transducer of Figure 4.3. Note that in the translation table for a tree transducer every variable which appears in the right side of a rule must appear in the left side, but not every variable which appears in the left side need be in the right side.

4.2. Formal Definition of a Tree Transducer within $L(\Theta_1)$ and $L(\Theta_2)$

To begin with, we provide the definition of a tree transducer (finite set of rules), which is just syntactic. Later on, in Chapter 6, we will consider translations which are determined by syntactic and semantic considerations. Thus, we will consider tree transducers which specify how to replace trees in the source data structure with semantically equivalent trees in the target data structure. The key problem is in finding the finite set of rules which correctly specify the translation.
Figure 4.3 Pictorial representation of tree transducer $T_2$. 

Row 1: Source tree: INSERT $X_1$ $X_2$ Target tree: CONS $X_1$ APPEND NIL $X_2$

Row 2: Source tree: INSERT $X_2$ INSERT $X_1$ Target tree: APPEND MAKE $X_1$ MAKE $X_2$

Row 3: Source tree: INSERT $b$ INSERT $2$ Target tree: CONS CONS $X_3$ 3 $X_3$
Figure 4.4 A tree transduction by tree transducer $T_2$
First, we provide the general definition of an algebraic tree transducer and then we give a definition for natural tree transducers, which can be built by using the least elements of $L(\Theta_1)$ and $L(\Theta_2)$.

4.2.1. Algebraic definition of a tree transducer

The algebraic tree transducer and other related concepts are defined below.

**Definition 4.3 [Signature Representation]**

$\gamma = (\Sigma', \Sigma, \eta, \pi)$ is a "signature representation" if $\Sigma'$ is represented in $\Sigma$ via representation functions $\eta$ and $\pi$.

**Definition 4.4 [Algebraic Tree Transducer]**

An "algebraic tree transducer" is a pair $T = (\gamma_1, \gamma_2)$ of signature representations $\gamma_i = (\Sigma', \Sigma_i, \eta_i, \pi_i)$ for $i = 1, 2$. The algebraic tree transducer determines a family of tree transductions $\{\text{TRANS}(T, s)/s \in N'\}$ where

$$\text{TRANS}(T, s) = \{(h_1(t), h_2(t))/t \in T_S^s, \quad h_1: T_{\Sigma_i} \to T_{\Sigma_i} \text{ for } i = 1, 2 \text{ is a unique homomorphism from } T_{\Sigma_i} \text{ to } T_{\Sigma_i}\}.$$ 

Furthermore, if $P \in \Sigma'$, $(\pi_1(P), \pi_2(P))$ is called a "row" of tree transducer $T$, and we write this as "$(\pi_1(P), \pi_2(P)) \in T$".

Thus, a tree transducer $(T)$ may be specified by homomorphisms $h_1$ and $h_2$ as illustrated in Figure 4.5. In this case, the tree transducer $T$ is given by the composition $h_2 \circ h_1^{-1}$. 
Figure 4.5 Informal view of a tree transducer
4.2.2. Example: formal construction of a tree transducer

**Example 4.5**

Consider Example 4.1 and Figure 4.2. In Example 4.1, we, informally, constructed two tree transducers $T_1$ and $T_2$. Now we make $T_1$ as a formal tree transducer with full specifications as in Figure 4.6. From now on, if the domains of each row of a tree transducer is understood from the context, then we will eliminate those domains; for example, the first row of the tree transducer in Figure 4.6 will be written as:

$$\text{INSERT}[X_1X_2] \xrightarrow{\pi_1} P_1: (R_1R_2) \xrightarrow{\pi_2} \text{CONS}[X_1X_2]$$

4.2.3. Tree transducer within $L(\Theta_1)$ and $L(\Theta_2)$

**Definition 4.6** [Tree Transducer within $L(\Theta_1)$ and $L(\Theta_2)$]

Let $\Theta_i \subseteq \Sigma_i$ be a finite set and $L_i$ be the least elements of $L(\Theta_i)$ for $i=1,2$, such that $L_1 = \{A_1,\ldots,A_n\}$ and $L_2 = \{B_1,\ldots,B_n\}$. If there exists signature representations $\gamma_i = (\Sigma_i^*,\pi_i^*,\eta_i^*,\pi_i)$ for $i=1,2$ such that $\Sigma_i^* = \{P_1,\ldots,P_n\}$ and for each $j=1,\ldots,n$, $\pi_i^*(P) \in \gamma_j$ and $\pi_2^*(P) \in \gamma_j$, then $T = (\gamma_1,\gamma_2)$ is a "tree transducer within $L(\Theta_1)$ and $L(\Theta_2)$." Note that by this definition if $P_i \neq P_j$, then $\pi_1^*(P_i) \neq \pi_1^*(P_j)$ and $\pi_2^*(P_i) \neq \pi_2^*(P_j)$.

Our definition of tree transducers within $L(\Theta_1)$ and $L(\Theta_2)$ differs from the definition of a tree transducers that has been developed in [31]. The difference is that we do not allow repeated source (target) trees in the tree transducer rows, while in [31] they...
\[ N' = \{ R_1, R_2 \} \]
\[ \Sigma'(R_1, R_2, R_2) = \{ P_1 \} \]
\[ \Sigma'(R_1, R_2, R_2, R_2) = \{ P_2, P_3 \} \]
\[ n_1(R_1) = D \]
\[ n_1(R_2) = S \]
\[ n_2(R_1) = A \]
\[ n_2(R_2) = L \]

1st row of \( T_1 \)
\[
\begin{cases}
\pi_1(P_1) = \text{INSERT}[X_1, X_2]^{DS} \\
\pi_2(P_1) = \text{CONS}[X_1, X_2]^{AL}
\end{cases}
\]

2nd row of \( T_1 \)
\[
\begin{cases}
\pi_1(P_2) = \text{INSERT}[X_1, \text{INSERT}[X_2, X_2]]^{DDS} \\
\pi_2(P_2) = \text{CONS}[X_1, \text{CONS}[X_2, \text{NIL}]]^{AAL}
\end{cases}
\]

3rd row of \( T_1 \)
\[
\begin{cases}
\pi_1(P_3) = \text{INSERT}[a, \text{INSERT}[b, X_3]]^{DDS} \\
\pi_2(P_3) = \text{CONS}[2, \text{CONS}[3, X_3]]^{AAL}
\end{cases}
\]

Figure 4.6 Full specification of tree transducer \( T_1 \)
permit this. That is, if $T$ is a tree transducer within $L(\theta_1)$ and $L(\theta_2)$ then $\pi_1$ and $\pi_2$ are injections from $\Sigma'$ into $\theta_1$ and $\theta_2$ respectively. Later on, we will remove this condition.

4.2.4. Mono tree transducer

Next, we define a restriction on tree transducers within $L(\theta_1)$ and $L(\theta_2)$ called "mono" tree transducers. These will be used in proving the major theorems of this chapter. The phrase "mono" is from the concept of "monotonicity" which is the property of a function when it gives larger values for larger arguments.

**Definition 4.7** [Mono Tree Transducer]

Let $T = (\gamma_1, \gamma_2)$ be a tree transducer within $L(\theta_1)$ and $L(\theta_2)$ which determines a family of tree transductions $\{\text{TRANS}(T, s)/s \in \mathbb{N}'\}$ where

$$\text{TRANS}(T, s) = \{(h_1(t), h_2(t))/t \in T_{\Sigma'}, h_i: T_{\Sigma'} \rightarrow T_{\Sigma'_i}, \text{ for } i = 1, 2\}$$

is a unique homomorphism from $T_{\Sigma'}$ to $T_{\Sigma'_i}$.

If $h_1(P) = P_1$, $h_1(q) = q_1$, and $\text{SIZE}(P_1) \geq \text{SIZE}(q_1)$ implies that $\text{SIZE}(P) \geq \text{SIZE}(q)$ then $T$ will be called a "source mono tree transducer". Similarly, if $h_2(P) = P_2$, $h_2(q) = q_2$, and $\text{SIZE}(P_2) \geq \text{SIZE}(q_2)$ implies that $\text{SIZE}(P) \geq \text{SIZE}(q)$ then $T$ will be called a "target mono tree transducer". Furthermore, $T$ is called a "mono tree transducer" if it is both source mono and a target mono tree transducer.
4.3. Introduction to Properties of Tree Transducers

As we shall see in Chapter 6, signature Σ, equations E, and tree transducers will be used for implementation of a source data structure by a target data structure. Properties of tree transducers (consistency, semiconsistency) within equations is going to be introduced for checking the implementation of a source data structure by a target data structure. That is, given a tree transducer T and sets of equations E₁ (source equations) and E₂ (target equations) we will examine whether each equation in E₁ is a "theorem" of the target language. In order to define properties of tree transducers, we need to define the following concepts.

4.3.1. Source tree and transduced tree

The following definition will be used in finding the output tree (transduced tree) given an input tree via a tree transducer.

**Definition 4.8**

Let \( \mathcal{T} = (\gamma_1, \gamma_2) \) be a tree transducer within \( \mathcal{L}(\Theta_1) \) and \( \mathcal{L}(\Theta_2) \) where \( \Theta_1 \subseteq \Sigma_1 \) and \( \gamma_i = (\Sigma'_i, \overline{\Sigma}_i, \eta_i, \pi_i) \) for \( i = 1, 2 \) and

\[
\text{TRANS}(\gamma, s) = \{(h_1(t), h_2(t))/t \in \Sigma'_s, \ h_i: \ T_{\Sigma'_i} \rightarrow T_{\overline{\Sigma}_i} \text{ for } i = 1, 2 \text{ is a unique homomorphism} \}.
\]

Let \( h_i \) be the derived homomorphism of \( h_i \) for \( i = 1, 2 \). Then we write \( f_1 \equiv f_2 \) if

1. there exists a \( f \in \Sigma' \), and
2. \( h_i(f_{\Sigma'}, [y_1, \ldots, y_k]) = f_i_{\Sigma'}([h_i(y_1), \ldots, h_i(y_k)]) \) for \( i = 1, 2 \). \( \square \)
4.3.2. Informal discussion of tree transduction

Definition 4.8 enables us to transform a nonterminated tree (tree with variable) into a nonterminated tree by using a tree transducer. Without having the Definition 4.8, we were just able to transform terminated trees into terminated trees. In order to see how the preceding definition works, we consider the following example.

Example 4.9

Let $T$ be a tree transducer with a single row such that:

\[ \pi_1(p) = b[X_1b[X_1X_2]]; \]
\[ \pi_2(p) = c[c[X_1X_2]X_2]; \text{ and } P \in \Sigma'(qq,q) \]

Let $f = P[P[X_1X_2]X_3]$ such that $f \in \Sigma'$, then $\tilde{h}_1(f_{\Sigma}, (y_1, y_2, y_3)) = b[b[b[X_1X_2]]b[b[X_1b[X_1X_2]]X_3]]_\Sigma'(h_1(y_1), h_1(y_2), h_1(y_3))$ and $\tilde{h}_2(f_{\Sigma}, (y_1, y_2, y_3)) = c[c[c[X_1X_2]X_2]X_3]_\Sigma'(h_2(y_1), h_2(y_2), h_2(y_3))$.

Hence, we can write

\[ \tilde{h}_1(f_{\Sigma}, (y_1, y_2, y_3)) = f_1(h_1(y_1), h_1(y_2), h_1(y_3)); \text{ and } \]
\[ \tilde{h}_2(f_{\Sigma}, (y_1, y_2, y_3)) = f_2(h_2(y_1), h_2(y_2), h_2(y_3)); \]

where

\[ f_1 = b[b[X_1b[X_1X_2]]b[b[X_1b[X_1X_2]]X_3]]; \]
\[ f_2 = c[c[c[X_1X_2]X_2]X_3]_X_3] \]

Thus $f_{\Sigma}'$. \qed
4.3.3. Informal discussion of consistency of tree transducers

We now present an example to illustrate the need for the concepts of "semiconsistency" and "consistency" of tree transducers. In the next section, we develop these ideas formally.

Example 4.10

The purpose of this example is to introduce the concept of semiconsistent and consistent tree transducers related with a set of equations.

Let $N_1 = N_2 = \{I\} \text{ and } \Sigma_i \text{ be an } N_i\text{-sorted signature and } E_i \text{ be a set of equations associated with } \Sigma_i \text{ for } i=1,2; \text{ such that }$

<table>
<thead>
<tr>
<th>signature $\Sigma_1$</th>
<th>signature $\Sigma_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma_1(II, I) = {+, \ast}$</td>
<td>$\Sigma_2(II, I) = {-, /}$</td>
</tr>
<tr>
<td>$\Sigma_1(\lambda, I) = {0,1} \text{ for } i=1,2$</td>
<td>$\Sigma_2(\lambda, I) = {0,1,\omega}$</td>
</tr>
</tbody>
</table>

$E_1$: source equations

(a1): $+[X_1X_2] = +[X_2X_1]$  
(a2): $+[X_1O] = X_1$  
(a3): $*[X_1X_2] = *[X_2X_1]$  
(a4): $*[0X_1] = 0$  
(a5): $*[1X_1] = X_1$  
(a6): $+[*[X_1X_2]*[X_1X_3]] = *[X_1+[X_2X_3]]$

$E_2$: target equations

(b1): $-[X_1O] = X_1$  
(b2): $+[\ast[X_1]] = X_1$  
(b3): $+[0] = 0$  
(b4): $-[0X_1] = \ast[X_1]$  
(b5): $-[X_1\ast[X_2]] = -[X_2\ast[X_1]]$  
(b6): $/[X_11] = X_1$
Now, let $\phi_1 = \{f_1, f_2, f_3, f_4, f_5, f_6\} \subseteq \mathcal{F}_1$, and $\phi_2 = \{f'_1, f'_2, f'_3, f'_4\}$ where

- $f_1 = +[X_1X_2]$
- $f_2 = +[X_2X_1]$
- $f_3 = *[X_1X_2]$
- $f_4 = *[X_2X_1]$
- $f_5 = 0$
- $f_6 = 1$

Then $L_1$ and $L_2$ are the least elements of $L(\phi_1)$ and $L(\phi_2)$ such that

- $L_1 = \{\{f_1, f_2\}, \{f_3, f_4\}, \{f_5\}, \{f_6\}\}$; and
- $L_2 = \{\{f'_1\}, \{f'_2\}, \{f'_3\}, \{f'_4\}\}$.

Figure 4.7 shows the pictorial representation of $L(\phi_1)$. Pictorial representation of $L(\phi_2)$ is similar to Figure 4.7 and will not be repeated here. Examining $L_1$ and $L_2$ will lead to the following tree transducer:

<table>
<thead>
<tr>
<th>row</th>
<th>source tree</th>
<th>target tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$+[X_1X_2]$</td>
<td>$-[X_1-[0X_2]]$</td>
</tr>
<tr>
<td>2</td>
<td>$*[X_1X_2]$</td>
<td>$/[X_1/[1X_2]]$</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 4.7. Pictorial representation of $L(0^1)$

$A = f_1 f_2$
$B = f_3 f_4$
$C = f_5$
$D = f_6$
Suppose that a source (target) data structure has been represented by $E_1$ and $E_1 (E_2$ and $E_2)$. In order to verify that tree transducer $T$ within $L(\phi_1)$ and $L(\phi_2)$ implements source data structure by target data structure, we have to prove that each equation of source data structure is a "theorem" of target data structure. That is, if $e = (L,R) \in E_1$, $L \equiv L'$, and $R \equiv R'$, then $[[L']] E_2 = [[R']] E_2$ must be true for every $e$ in $E_1$.

Let us show how to prove $+\{X_1 X_2\} = +\{X_2 X_1\}$, regarding the fact that target data structure has already been implemented. Using $A = -\{X_1 -\{0 X_2\}\}$ and $B = -\{X_2 -\{0 X_1\}\}$:

$+\{X_1 X_2\} \not\equiv A$

$+\{X_2 X_1\} \not\equiv B$

Now, we have to show that $[[A]] E_2 = [[B]] E_2$,

$[[A]] E_2 = \{-[X_1^{-}[X_2]]\}, \{-[X_2^{-}[X_1]]\}$,

$[[B]] E_2 = \{-[X_2^{-}[X_1]]\}, \{-[X_1^{-}[X_2]]\}$.

Thus, $[[A]] E_2 = [[B]] E_2$ and this implies that properties of source data structure (equations $E_1$) can be established by using properties (operations) of target data structure.

Also, to prove $*\{0 X_1\} = 0$, we have $*\{0 X_1\} \not\equiv /\{0 /\{1 X_1\}\}$ and $0 \not\equiv 0$

So, we have to show that $[[0]] E_2 = [[/\{0 /\{1 X_1\}\}]] E_2$

$[[0]] E_2 = \{0\}$

$[[/\{0 /\{1 X_1\}\}]] E_2 = \{0\}$ since $/\{0 /\{1 X_1\}\} \not\equiv 0$.

Verifying other equations of $E_1$ is left to the reader.
4.4. Formal Definition of Consistency and Semiconsistency

In this part, we will be concerned with the notion of a "consistent" and "semiconsistent" tree transducers. These concepts will be used in Chapter 6 for the purpose of implementing a source data structure by target data structure. Before defining semiconsistency and consistency of a tree transducer, we first present the following example.

4.4.1. Example: Boolean type implementation

Example 4.11

This example implements the OR function by AND and NOT functions.

Consider $\Sigma_i$ and $E_i$ for $i=1,2$.

**$\Sigma_1$: source signature**

$\Sigma_1(\lambda, B) = \{T,F\}$

$\Sigma_1(B, B) = \{\text{NOT}\}$

$\Sigma_1(BB, B) = \{\text{OR, IMPLIES}\}$

**$\Sigma_2$: target signature**

$\Sigma_2(\lambda, B) = \{T,F\}$

$\Sigma_2(B, B) = \{\text{NOT}\}$

$\Sigma_2(BB, B) = \{\text{AND}\}$

**$E_1$: source equations**

\begin{align*}
(e1): & \quad \text{OR}[X_1X_2] = \text{OR}[X_2X_1] \\
(e2): & \quad \text{OR}[TX_1] = T \\
(e3): & \quad \text{OR}[FX_1] = X_1 \\
(e4): & \quad \text{IMPLIES}[X_1X_2] = \text{OR}[\text{NOT}[X_1X_2]]
\end{align*}

**$E_2$: target equations**

\begin{align*}
(c1): & \quad \text{NOT}[F] = T \\
(c2): & \quad \text{NOT}[T] = F \\
(c3): & \quad \text{NOT}[\text{NOT}[X_1]] = X_1 \\
(c4): & \quad \text{AND}[X_1X_2] = \text{AND}[X_2X_1] \\
(c5): & \quad \text{AND}[TX_1] = X_1 \\
(c6): & \quad \text{AND}[FX_1] = F
\end{align*}
Furthermore, let $T$ (Figure 4.8) be a tree transducer within $L(\theta_1)$ and $L(\theta_2)$ for some $\theta_1$ and $\theta_2$. Now, we show that OR function can be implemented by the tree transducer $T$ and using the target equations. Hence, we have to show that equations $e_1$, $e_2$, $e_3$, and $e_4$ are "theorems" of the target language. That is, if $e = (L,R)eE_1$, $L \rightarrow L'$, and $R \rightarrow R'$ then $[[L']]_{E_2} = [[R']]_{E_2}$ must be true.

To prove $OR[X_1X_2] = OR[X_2X_1]$

using $A = NOT[AND[NOT[X_1]NOT[X_2]]]$; and

$B = NOT[AND[NOT[X_2]NOT[X_1]]]$; and

using tree transducer $T$ we have:

$OR[X_1X_2]T^A$ and $OR[X_2X_1]T^B$, $[[A]]_{E_2} = \{A,B\}$ and $[[B]]_{E_2} = \{B,A\}$.

Thus, $[[A]]_{E_2} = [[B]]_{E_2}$ implies that $OR[X_1X_2] = OR[X_2X_1]$. Also, in order to prove $OR[TX_1] = T$, using $A_1 = NOT[AND[NOT[T]NOT[X_1]]]$ and $B_1 = T$ we have: $OR[TX_1]T^{A_1}$ and $T^{B_1}$, $[[B_1]]_{E_2} = \{T\}$.

$A_1 \Rightarrow NOT[AND[F NOT[X_1]]]$

$\Rightarrow NOT[F]$

$\Rightarrow T$

Thus, $[[A_1]]_{E_2} = [[B_1]]_{E_2} = \{T\}$.

To prove $OR[FX_1] = X_1$, using $A_2 = NOT[AND[NOT[F]NOT[X_1]]]$ and $B_2 = X_1$, and using $T$ we have:
\[ N' = \{ M \}, \quad n_1(M) = n_2(M) = B \]
\[ \Sigma' = \{ \Sigma'(w, n)/w \in \mathbb{M}^*, \quad n \in \mathbb{M} \} \]
\[ = \{ P_1, P_2, P_3, P_4, P_5, P_6 \} \]

1st row of \( T \)
\[
\begin{align*}
\pi_1(P_1) &= \text{OR}[X_1X_2] \\
\pi_2(P_1) &= \text{NOT[AND[NOT[X_1]NOT[X_2]]]} \\
\end{align*}
\]

2nd row of \( T \)
\[
\begin{align*}
\pi_1(P_2) &= \text{OR}[\text{NOT}[X_1]\text{NOT}[X_2]] \\
\pi_2(P_2) &= \text{NOT[AND[X_1X_2]]} \\
\end{align*}
\]

3rd row of \( T \)
\[
\begin{align*}
\pi_1(P_3) &= \text{NOT[OR[X_1X_2]]} \\
\pi_2(P_3) &= \text{AND[NOT[X_1]NOT[X_2]]} \\
\end{align*}
\]

4th row of \( T \)
\[
\begin{align*}
\pi_1(P_4) &= \text{IMPLIES}[X_1X_2] \\
\pi_2(P_4) &= \text{NOT[AND[X_1NOT[X_2]]]} \\
\end{align*}
\]

5th row of \( T \)
\[
\begin{align*}
\pi_1(P_5) &= T \\
\pi_2(P_5) &= T \\
\end{align*}
\]

6th row of \( T \)
\[
\begin{align*}
\pi_1(P_6) &= F \\
\pi_2(P_6) &= F \\
\end{align*}
\]

Figure 4.8 Tree transducer \( T \) within \( L(\Theta_1) \) and \( L(\Theta_2) \)
\( \text{OR}(F, X_1) \Rightarrow A_2 \) and \( X_1 \Rightarrow B_2 \) where \( [[B_2]]_{E_2} = \{X_1\} \)

\( A_2 \Rightarrow \text{NOT}(\text{AND}(T, \text{NOT}[X_1])) \)

\( E_2 \Rightarrow \text{NOT}([X_1]) \)

\( E_2 \Rightarrow X_1 \)

Thus, \( [[A_2]]_{E_2} = \{X_1\} \), which implies \( [[A_2]]_{E_2} = [[B_2]]_{E_2} = \{X_1\} \).

Similarly, it can be proved that equation e4 is a theorem. Thus, we prove that the source language (having operations OR, NOT, IMPLIES) can be implemented by the target language (having operations NOT, AND).

We are now in a position to study the properties of tree transducers with respect the source and target equations. That is, to investigate relationships between tree transducers and equations. We will consider two kinds of tree transducers. First, we define "local consistency" and "local semiconsistency". Then, we define "global consistency" and "global semiconsistency".

4.4.2. Local consistency and local semiconsistency

Definition 4.12 [Local Consistency and Local Semiconsistency]

Let \( T = (Y_1, Y_2) \) be a tree transducer within \( L(\Theta_1) \) and \( L(\Theta_2) \).

Let \( E_1 \) be a set of source equations and \( E_2 \) a set of target equations associated with \( E_1 \) and \( E_2 \) respectively. Then \( T \) is said to be "locally consistent with respect to \( E_1 \) and \( E_2 \)" if conditions (CL1) and
(CL2) are satisfied. Also, T is said to be "locally semiconsistent" from source (target) to target (source) if CL1 (CL2) is satisfied.

(CL1): for all \((L=R)\in E_1\),
\[L' \supset L \text{ and } R' \supset R \implies [L'] E_2 = [R'] E_2\]

(CL2): for all \((L=R)\in E_2\)
\[L' \supset L \text{ and } R' \supset R \implies [L'] E_1 = [R'] E_1\]

Furthermore, if T is "locally consistent with respect to \(E_1\) and \(E_2\)" and \(E_1, E_2\) are understood from the context then we will leave out \(E_1\) and \(E_2\).

4.4.3. Global consistency and global semiconsistency

Next, regarding the definition of local consistency and local semiconsistency, we define "global consistency" and "global semi-consistency".

**Definition 4.13** [Global Consistency and Global Semiconsistency]

Let \(T = (\gamma_1, \gamma_2)\) be a tree transducer within \(L(\Sigma_1)\) and \(L(\Sigma_2)\). Let \(E_1\) be a set of source equations and \(E_2\) be a set of target equations associated with \(\Sigma_1\) and \(\Sigma_2\) respectively. Then T is said to be "globally consistent with respect to \(E_1\) and \(E_2\)" if the following conditions (CG1) and (CG2) are satisfied. Also, T is said to be "globally semiconsistent" from source (target) to target (source) if CG1(CG2) is satisfied.

(CG1): If \([s] E_1 = [t] E_1, s \supset s', \text{ and } t \supset t'\) then \([s'] E_2 = [t'] E_2\).
(CG2): If \([s]_{E_2} = \left[t\right]_{E_2} \cdot s^{\pi_1}, \text{ and } t^{\pi_2} \text{ then } [s]_{E_1} = [t]_{E_1}\]

Consistency (semiconsistency) differs from the previous approaches [15, 39] for correctness of implementation (Chapter 6) of one data structure \((\Sigma_1 \text{ along with } E_1)\) by another \((\Sigma_2 \text{ along with } E_2)\). In [15, 39] for demonstrating the correctness of an implementation of source data structure with respect to the target data structure, they establish a homomorphism from the source language to the target language. If \(T\) is "globally consistent with respect to \(E_1\) and \(E_2\)" and \(E_1, E_2\) are understood from the context, then we will call \(T\) as a "globally consistent" tree transducer.

4.5. Weak Syntactical Honest Tree Transducer

In this section, we will identify another important property of tree transducers by using algebraic equations. Then, this property will be used to prove two important theorems. These concepts will be generalized in Chapter 6.

Definition 4.14  [Weak Syntactical Honest Tree Transducer]

Let \(T = (\gamma_1, \gamma_2)\) be a tree transducer within \(L(\Theta_1)\) and \(L(\Theta_2)\) such that 
\[
\gamma_i = (\Sigma_i', \Sigma_i, \Sigma_i, \eta_i, h_i) \text{ for } i=1,2 \text{ and } \text{TRANS}(T, a) = \{(h_1(w), h_2(w))/h_i: \Sigma_i \rightarrow^* \Sigma_i', \text{ and } h_i \text{ is a unique homomorphism for } i=1,2\}.
\]

Let \(E_1(E_2)\) be a set of source (target) equations. Tree transducer \(T\) is "weak syntactical honest" if
i) \( T \) is locally semiconsistent from source to target;

ii) \( E_1 \) and \( E_2 \) are non-expanding;

iii) \( T \) is mono from source to target;

iv) \( \text{SBASE} = \{h_1(w) / |w| \leq 2\}, \text{TBASE} = \{h_2(w) / |w| \leq 2\} \) if

\[ \text{SB} \in \text{the least element of } L(\text{SBASE}), b_1, b_2 \in \text{SB}, b_i \neq c_i \]

\[ c_i \neq d_i \text{ for } i=1,2 \] implies that there exists a \( TB \) in the least element of \( L(\text{TBASE}) \) and \( d_1, d_2 \in TB \)

v) \[ \llbracket \pi_1(p) \rrbracket_{E_1} = \llbracket \pi_2(q) \rrbracket_{E_2} \text{ implies } \llbracket \pi_2(p) \rrbracket_{E_2} = \llbracket \pi_2(q) \rrbracket_{E_2} \]

where \( p, q \in \Sigma' \)

\[ \square \]

**Definition 4.15 [Weak SYNIMPL]**

Let \( T = (Y_1, Y_2) \) be a tree transducer. Let \( E_1(E_2) \) be source (target) equations. If \( A \xrightarrow{E_1} B, A \xrightarrow{A'} B' \) implies that

\[ \llbracket A' \rrbracket_{E_2} = \llbracket B' \rrbracket_{E_2} \]

then \( T \) will be called as a "weak SYNIMPL " tree transducer.

In Chapter 6, we will generalize this definition. That is, we will define a SYNIMPL (syntactical implementation) tree transducer, that is a generalization of the above definition. In the above definition we require \( \llbracket A' \rrbracket_{E_2} = \llbracket B' \rrbracket_{E_2} \) if \( A \) transforms to \( A' \), \( B \) to \( B' \) and \( A \xrightarrow{E_1} B \). A SYNIMPL tree transducer will have

\[ \llbracket A' \rrbracket_{E_2} = \llbracket B' \rrbracket_{E_2} \text{ if } \llbracket A \rrbracket_{E_1} = \llbracket B \rrbracket_{E_1} \]

Thus, a weak SYNIMPL tree transducer is a special case of the more general SYNIMPL tree transducer studied later.
4.6. Major Theorems

Theorem 4.16

If T is a weak syntactical honest tree transducer, and \( E_1(E_2) \) are a set of source (target) equations, then T is a weak SYNIMPL tree transducer.

Proof: Using Definition 3.3, we consider three cases and we have to show that if \( A \xrightarrow{T} B, A \xrightarrow{T} C, B \xrightarrow{T} D \), then \([C]_E_2 = [D]_E_2\).

Case (i)

If \( A = L, B = R, \) and \( (L,R) \in E_1 \). Let \( L' = L \) and \( R' = R' \). Since T is honest, then T must be locally semiconsistent from source to target; and this implies that \([L']_E_2 = [R']_E_2\). Now, setting \( L' = C \) and \( R' = D \), we conclude that \([C]_E_2 = [D]_E_2\).

Case (ii)

If \( A = L \equiv \{t_1, \ldots, t_n\} \) and \( B = R \equiv \{t_1, \ldots, t_n\} \), and \( (L,R) \in E_1 \). Let \( A \xrightarrow{T} C \) and \( B \xrightarrow{T} D \), then we must have \( C = L'[t_1', \ldots, t_n'] \) and \( D = R'[t_1', \ldots, t_n'] \). Since T is honest, thus, T is locally semiconsistent from source to target. Hence \((L,R) \in E_1, L \xrightarrow{T} L' \) and \( R \xrightarrow{T} R' \) implies \([L']_E_2 = [R']_E_2\).

Using lemma 3.18, we must have \([C]_E_2 = [D]_E_2\).

Case (iii)

If \( A = f(t_1, \ldots, t_j, \ldots, t_n) \),
\( B = f(t_1, \ldots, t_j, \ldots, t_n) \), and
\( t_j \xrightarrow{e} t_j' \) and \( e \in (L,R) \in E_1 \).
Let $A \subseteq C$ and $B \subseteq D$. For proving $[[C]]_{E_2} = [[D]]_{E_2}$, we will use induction on $|w|$ where $h_1(w) = A$.

**base step:** $1 \leq |w| \leq 2$

Using condition (iv) of Definition 4.14. Letting $b_1 = c_1 = A$, $b_2 = A$, $c_2 = B$, and $b_1, b_2 \in SB$, an element of the least element of $L(SBASE)$ where $SBASE = \{h_1(w)/|w| \leq 2\}$. Condition (iv) of Definition 4.14 implies that $d_1 = C$ and $d_2 = D$ and $d_1, d_2 \in TB$, an element of the least element of $L(TBASE)$ where $TBASE = \{h_j(w)/|w| \leq 2\}$. Using Theorem 3.41, we must have $[[d_1]]_{E_2} = [[d_2]]_{E_2}$. Thus, we must have $[[C]]_{E_2} = [[D]]_{E_2}$.

**inductive step:** Given $h_1(w) = f_1, f_1 \geq f_2, |w| = m, f_1 \geq f_1', f_2 \geq f_2'$ which implies $[[f_1]]_{E_2} = [[f_2]]_{E_2}$. We have to show that if $h_1(w) = A, A \geq B, |w| = m+1, A \subseteq C, B \subseteq D$, then $[[C]]_{E_2} = [[D]]_{E_2}$.

**case (1)**

Let $A = \pi_1(P)(h_1(w_1), \ldots, h_1(w_r))$ (1)

$\pi_1(P) = f(\ldots, t_j, \ldots)$ (2)

$B = \pi_1(Q)(h_1(w_1), \ldots, h_1(w_r))$ (3)

$\pi_1(Q) = f(\ldots, t_j', \ldots)$ (4)

$[[\pi_1(P)]]_{E_1} = [[\pi_1(Q)]]_{E_1}$ (5)

Thus,

$[[C]]_{E_2} = [[\pi_2(P)(h_2(w_1), \ldots, h_2(w_r))]_{E_2}$

$= [[\pi_2(Q)(h_2(w_1), \ldots, h_2(w_r))]_{E_2}$ from (5)

$= [[D]]_{E_2}$. 

case (2)

\[ A = \pi_1(P)(h_1(w_1), \ldots, h_1(w'), \ldots, h_1(w_r)) \]

\[ B = \pi_1(q)(h_1(w_1), \ldots, h_1(w''), \ldots, h_1(w_r)) \]

such that \( t_j \) is a subtree of \( h_1(w') \) and \( t'_j \) is a subtree of \( h_1(w'') \), \( j \geq i \) and \( h_1(w') \Rightarrow h_1(w'') \), and \( \pi_1(P) = \pi_1(q) \).

\[ C = \pi_2(P)(h_2(w_1), \ldots, h_2(w'), \ldots, h_2(w_r)) \]

\[ D = \pi_2(q)(h_2(w_1), \ldots, h_2(w''), \ldots, h_2(w_r)) \]

\( \pi_1(P) = \pi_1(q) \) implies \( \llbracket \pi_2(P) \rrbracket_{E_2} = \llbracket \pi_2(q) \rrbracket_{E_2} \)

\[ \llbracket C \rrbracket_{E_2} = \llbracket \pi_2(P)(h_2(w_1), \ldots, h_2(w'), \ldots, h_2(w_r)) \rrbracket_{E_2} \]

\[ = \llbracket \pi_2(P)(h_2(w_1), \ldots, x, \ldots, h_2(w_r)) \rrbracket_{E_2} \]

\[ h_2(w') \Rightarrow x \text{ by induction} \]

\[ = \llbracket \pi_2(P)(h_2(w_1), \ldots, h_2(w''), \ldots, h_2(w_r)) \rrbracket_{E_2} \]

\[ h_2(w'') \Rightarrow x \text{ by induction} \]

This theorem proved that under certain conditions if \( A \nearrow B, A \Rightarrow C, B \Rightarrow D \) then \( C \equiv D \).

In Chapter 6, we will generalize this and will show (under special conditions) that if \( \llbracket A \rrbracket_{E_1} = \llbracket B \rrbracket_{E_1}, A \Rightarrow C, B \Rightarrow D \), then \( \llbracket C \rrbracket_{E_2} = \llbracket D \rrbracket_{E_2} \).

**Theorem 4.17**

Let \( T \) be a weak syntactical honest tree transducer. Let \( s \equiv s', \]

\( s \equiv s', \llbracket s \rrbracket_{E_2} = \llbracket s' \rrbracket_{E_2} \), and \( t \not\equiv s \). If \( t \equiv t_1 \) and \( t \equiv t_2 \) then \( \llbracket t_1 \rrbracket_{E_2} = \llbracket t_2 \rrbracket_{E_2} \).

**Proof:** \( t \not\equiv s \) implies that \( t \Rightarrow A_1 \Rightarrow A_2 \Rightarrow \ldots \Rightarrow A_n \Rightarrow s \)
Now let $A_i \to A'_i$ and $A_i \to A''_i$ for $i=1,...,n$. Using Theorem 4.16, we conclude that $\| A'_i \|_{E_2} = \| A''_{i+1} \|_{E_2}$ for $i=1,...,n-1$ (*1)

but we know that $s_1 = A'_n$ and $s_2 = A''_n$, thus,

$$\| A'_n \|_{E_2} = \| A''_n \|_{E_2}$$

(*2)

Comparing (*1) and (*2), we conclude that $\| t_1 \|_{E_2} = \| t_2 \|_{E_2}$. □

Theorem 4.17 generalizes Theorem 4.16 by requiring that $t \mathrel{\not\in}_{E_1} s$ in $n$ steps where $n \geq 1$ (Theorem 4.16 did this only for $n=1$). This is getting closer to what we actually desire. In general, we would like a theorem like 4.17 to hold if $t$ and $s$ were equivalent, i.e., if $\| t \|_{E_1} = \| s \|_{E_1}$. This generalization, however, will require some additional concepts and conditions, and is developed in Chapter 6.

4.7. Simplifications, Summary

We have examined the problem of constructing an algebraic tree transducers using the concept of lattice theory. Then, we identified certain important properties of tree transducers (consistency and semi-consistency) within the source and target equations. We then developed the concept of "weak syntactical honest" and "weak SYNIMPL" tree transducers. Also, we proved that if a tree transducer ($T$) is weakly syntactical honest then $T$ is a weak SYNIMPL. The concepts of this chapter will play an important role in the development of Chapter 6.
5. CONDITIONAL REWRITING SYSTEM

The definition of some data structures (signature $\Sigma$ along with equations $E$) requires the use of conditional equations (rewrite rules), that is, equations which do not hold for all possible substitutions of expressions for the variables, but which hold only for substitutions which satisfy some conditions.

5.1. The Idea of Conditional Rewrite Rules

In this chapter, we investigate conditional rewriting rules (conditional equations). One of the important implications of conditional rewriting rules is that it provides a powerful machinery for expressing the properties of complex data structures. By using simple equations, equations discussed earlier, we will only be able to work with fairly simple data structures. In general, it will be shown that simple equations are not sufficient for expressing the properties of data structures. Indeed, there exist data structures (array, set, ...) which have finite conditional specifications but only infinite equational specifications.

5.2. Related Work and Discussion

Simple equations and conditional equations for expressing the properties of data structures (signature $\Sigma$ along with equations $E$) can be seen in Guttag et al. [16], Goguen [9], Huet and Oppen [23], Thatcher et al. [41], and Liskov and Zilles [34]. Some of the researchers used conditional equations without distinguishing between equations and conditional equations.
In [41], they argue that using conditional equations allows powerful deductive techniques for establishing correctness of implementation of data structures. Also, in [41] they provide examples demonstrating that conditional equations are powerful than simple equations. They give a data structure whose properties can not be expressed by simple equations but can be expressed by conditional equations or by an infinite number of equations.

The approach taken by [16] and [14] involves the introduction of an auxiliary function "if-then-else". They have provided sufficient conditions for termination of equations using the if-then-else. Guttag et al. [16] and Thatcher et al. [42] also have focused on using specification of data structures with using if-then-else function.

Our research formalizes and provides a notion of conditional equations which differs from the above works. Our approach handles the if-then-else concept using Horn clauses.

5.3. Conditional Equations versus Simple Equations

The purpose of this section is two fold. First, we introduce an example, which shows the necessity of introducing the "conditional equations" and shows the drawbacks of simple equations. Then, after introducing the concept of equivalence data structures (Chapter 6), we will prove that the "conditional equations" are "more powerful" than simple equations.
Example 5.1

This example considers conditional equations in a data structure context. Consider the integer set data structure SET (signature Σ along with a set of equations), which has operations to create a new set by adding an integer to an existing set, to create a set by removing an integer from an existing set, to find the largest integer value of existing set, and to test if an integer is in a set. These operations are defined over the set of Integers, the set of Booleans, and the set of integer sets. The functionality of operations is an N-sorted signature Σ such that $N=\{\text{Set, Integer, Boolean}\}=\{S,I,B\}$; and

$\Sigma(S,I) = \{\text{LARGE}\}$

$\Sigma(\lambda,B) = \{\text{TRUE, FALSE}\}$

$\Sigma(\lambda,I) = \{0, \text{UNDEFINED}\}$

$\Sigma(S,B) = \{\text{ISEMPTY}\}$

$\Sigma(I,I) = \{\text{SUCC}\}$

$\Sigma(S,I) = \{\text{MEMBER}\}$

$\Sigma(SI,S) = \{\text{INSERT, DELETE}\}$

$\Sigma(SI,B) = \{\text{MEMBER}\}$

$\Sigma(\lambda,S) = \{\emptyset\}$

Let s be a variable which range over integer set objects (Sets) and i,j be variables which range over the Integers. The operation $\emptyset$ is the empty set, operation $\text{INSERT}(s,i)$ means that insert i into set s, $\text{DELETE}(s,i)$ indicates that the integer i must be deleted from the set s, and operation $\text{LARGE}(s)$ yields the largest integer in set s. For example, $\text{DELETE}(\text{INSERT}(s,i),j)$ is observably equivalent to $\text{INSERT}(\text{DELETE}(s,j),i)$ only if i and j are not equal. Also, we assume that in the SET data structure outlined above, adding the same integer twice should have no observable effect. This property can be expressed
by the equation:

\[(i=j) \Rightarrow \text{INSERT(INSERT}(s,i),j) = \text{INSERT}(s,i) \] (*1)

where "\Rightarrow" stands for implication. Equation (*1) is a "conditional equation" and indicates that \(\text{INSERT(INSERT}(s,i),j)\) can be replaced by \(\text{INSERT}(s,i)\) only if \(i=j\). Also, let the order of adding integer be unimportant to the SET data structure, which can be expressed by the equation:

\[\text{INSERT(INSERT}(s,i),j) = \text{INSERT}(\text{INSERT}(s,j),i) \] (*2)

Thus, equations (*1) and (*2) can be used and combined to establish equivalences among elements of SET data structure. Informally, the equations (E) for SET data structure has shown in Figure 5.1. We should note that every expression in the integer Set operations \(\text{T}_S\) is equivalent to an expression of the form:

\[\text{INSERT}(...)\text{INSERT}(∅,i_1)>...i_m>\]

where \(i_1<i_2<...<i_m\).

In Figure 5.1, equations (2), (3), (4), (8), (9), (13), and (14) are conditional equations. Thus, a conditional equation is a tuple \(\langle b,e \rangle\) where \(b\) is a coolean condition and \(e=\langle L,R \rangle\) is a simple equation, which is called the consequent of the conditional equation. Conditional equation \(\langle b,\langle L,R \rangle \rangle\) is like a law but the equality of expressions \(L\) and \(R\) does not always hold. That is, whenever Boolean condition \(b\) holds then \(L\) can be replaced by \(R\). \(\Box\)
s is a variable over Sets
i and j are variables over Integers

(1): DELETE (Ø,i) = Ø
(2): (i=j)→DELETE(INSERT(s,i),j) = DELETE(s,i)
(3): (i≠j)→DELETE(INSERT(s,i),j) = INSERT(DELETE(s,j),i)
(4): (i=j)→INSERT(INSERT(s,i),j) = INSERT(s,i)
(5): INSERT(INSERT(s,i),j) = INSERT(INSERT(s,j),i)
(6): LARGE(Ø) = UNDEFINED
(7): LARGE(INSERT(Ø,i)) = i
(8): (i<j)→LARGE(INSERT(INSERT(s,i),j)) = LARGE(INSERT(s,j))
(9): (i≥j)→LARGE(INSERT(INSERT(s,i),j)) = LARGE(INSERT(s,i))
(10): ISEMPTY(Ø) = TRUE
(11): ISEMPTY(INSERT(s,i)) = FALSE
(12): MEMBER(Ø,i) = FALSE
(13): (i=j)→MEMBER(INSERT(s,i),j)) = TRUE
(14): (i≠j)→MEMBER(INSERT(s,i),j)) = MEMBER(s,j)

Figure 5.1 Equations for SET data structure
5.4. Formal Definition of Conditional Equations

By studying Example 5.1 we observe that, it is possible to include conditional equations in specification of data structures. The benefit of conditional equations is that they yield more precise specifications for expressing the properties of complex data structures. First, we give the definition of "\(\Sigma\)-Boolean condition", which will be used in formalizing the definition of "conditional equations".

**Definition 5.2 [\(\Sigma\)-Boolean Condition]**

Let \(\Sigma\) be an \(N\)-sorted signature. Then "\(\Sigma(w,s)\)-Boolean conditions" are defined as

1) TRUE and FALSE are \(\Sigma(w,s)\)-Boolean conditions.
2) if \(b_1, b_2 \in \Sigma(w,s)\) then \((b_1 \equiv b_2)\) is a \(\Sigma(w,s)\)-Boolean condition.
3) if \(b_1, b_2, \text{ and } b\) are \(\Sigma(w,s)\)-Boolean conditions then \((b_1 \text{ and } b_2), (b_1 \text{ or } b_2), \text{ and } (\text{not } b)\) are \(\Sigma(w,s)\)-Boolean conditions.

Furthermore, we write "\(\Sigma\)-Boolean condition" (or just "\(\Sigma\)-Boolean") instead of "\(\Sigma(w,s)\)-Boolean condition" whenever \(w\) and \(s\) are understood from the context.

We will be mostly interested in \(\Sigma\)-Boolean conditions. Now, let us see how to evaluate \(\Sigma\)-Boolean conditions.

**Definition 5.3 [Evaluation of a \(\Sigma(w,s)\)-Boolean Condition]**

Let \(b\) be a \(\Sigma(w,s)\)-Boolean condition and let the symbols \(\neg\), \(\vee\) and \(\&\) respectively used for negation, disjunction (or), and con-
junction (and). Let \( A \) be a \( \Sigma \)-algebra. Then the evaluation of \( b_A \) is defined using a function \( \text{EVAL} \) as below:

1) if \( b_A^\prime = \text{TRUE} \) or \( b_A^\prime = \text{FALSE} \) then \( \text{EVAL}(b_A^\prime) = b_A^\prime \).

2) if \( f', f'' \in \Sigma(w, s), y_i \in A_i \) for \( i = 1, \ldots, n \), \( w = s_1 \ldots s_n \), and \( b = (f' = f'') \) then

\[
\text{EVAL}(b_A(y_1, \ldots, y_n)) = \begin{cases} 
\text{TRUE} & \text{if } f'_A(y_1, \ldots, y_n) = f''_A(y_1, \ldots, y_n) \\
\text{FALSE} & \text{otherwise}
\end{cases}
\]

Sometimes, we write \( \text{EVAL}(b_A^\prime) \) for \( \text{EVAL}(b_A^\prime(y_1, \ldots, y_n)) \) when \( y_1, \ldots, y_n \) are understood.

3) if \( b_A = (b_A^\prime \text{ and } b_A^\prime') \), then \( \text{EVAL}(b_A) = \text{EVAL}(b_A^\prime) \& \text{EVAL}(b_A') \).

4) if \( b_A = (b_A^\prime \text{ or } b_A^\prime') \) then \( \text{EVAL}(b_A) = \text{EVAL}(b_A^\prime) \lor \text{EVAL}(b_A') \).

5) if \( b_A = (\text{not } b_A^\prime) \) then \( \text{EVAL}(b_A) = \neg \text{EVAL}(b_A^\prime) \).

For the preceding definition, if algebra \( A \) is understood, then we will leave out the subscripts of \( A \).

**Definition 5.4 [Conditional Equations]**

Let \( \Sigma \) be an \( N \)-sorted signature. A "conditional equation" is a tuple \( c = <b, e> \) where \( b \) is a \( \Sigma(w, s) \)-boolean condition and \( e \) is a simple equation of the form \( e = <L, R> \in \Sigma(w, s) \times \Sigma(w, s) \) as discussed earlier. If \( c = <b, e> \) is a conditional equation then \( e \) is called the "consequent" of \( c \) and we write \( <b, e> \) in the notation.

\( c: b \Rightarrow e \)

Furthermore, if \( A \) is a \( \Sigma \)-algebra, then \( A \) "satisfies" \( <b, e> \) if and only if for all \( (y_1, \ldots, y_n) \in A_{s_1} \times \cdots \times A_{s_n} \), \( \text{EVAL}(b_A(y_1, \ldots, y_n)) = \text{TRUE} \) implies that \( L_A(y_1, \ldots, y_n) = R_A(y_1, \ldots, y_n) \). A \( \Sigma \)-algebra \( A \) "satisfies"
CE (a set of conditional equations) if and only if A "satisfies" every c in CE, and is called a \((\Sigma, CE)\)-algebra, and the class of all \((\Sigma, CE)\)-algebras with all homomorphisms between them is denoted \(\text{ALG}(\Sigma, CE)\).

A signature \(\Sigma\) along with CE determines the class of \(\text{ALG}(\Sigma, CE)\) of \(\Sigma\)-algebras satisfying CE. The class of \(\text{ALG}(\Sigma, CE)\) of \(\Sigma\)-algebras satisfying a family CE of equations has an "initial algebra" which is obtained as a "quotient of \(T_\Sigma\) by the smallest equivalences satisfying CE" \([41, 42]\). Also, we should note that conditional equations include the definition of simple equations. That is, if \(<L, R>\) is a simple equation then it can be replaced by a conditional equation such as \(<\text{TRUE}, <L, R>>\). That is, regardless of any condition \(L\) can be replaced by \(R\).

**Definition 5.5** [Application of CE to \(\Sigma\)]

Let \(\Sigma\) be an \(N\)-sorted signature and let \(CE=\{c_1, \ldots, c_n\}\) be a set of conditional equations. Let \(c=<b, <L, R>>\) be in CE. Then \(\Rightarrow\) is a relation defined as

(i) if \(f_1=L, f_2=R,\) and \(\text{EVAL}(b_\Sigma)=\text{TRUE}\) then \(f_1 \Rightarrow f_2\).

(ii) if \(f'=L_\Sigma(t_1, \ldots, t_m), f''=R_\Sigma(t_1, \ldots, t_m),\) and \(\text{EVAL}(b_\Sigma(t_1, \ldots, t_m))=\text{TRUE},\) then \(f' \Rightarrow f''\).

(iii) if \(f_1=f_\Sigma(y_1, \ldots, y_j, \ldots, y_m), f_2=f_\Sigma(y_1, \ldots, y'_j, \ldots, t_m),\) and \(y_j \Rightarrow y'_j\) then \(f_1 \Rightarrow f_2\).
Furthermore, $f = \{f'/f \Rightarrow f'\}, \quad f^C = \{f'/f \Rightarrow f' \text{ for every } c \in CE\}$, and if $f' \in f^C$, then we write $f \Rightarrow f'$.

If $<b,<L,R>>$ is a conditional equation, then $<L,R>$ will be treated as a directed rewrite rule, as discussed earlier. For a conditional equation $c$, the relation $\Rightarrow_c$ will be called a "reduction" relation. Also, if $CE$ is a set of conditional equations, then we denote the reflexive and transitive closure of $\Rightarrow$ by $\Rightarrow^*_C$ and $\Rightarrow^+_C$ respectively. Likewise, $\Rightarrow^*_C$, $\Rightarrow^+_C$, and $\Rightarrow^*_{CE}$ denote the complements of these relations. The subscript $CE$ will be left out when understood from the context. Also, from now on the terms "conditional equations", "conditional rewriting system", and "conditional rewrite rules" will be used equivalently.

Next, we define the "cycle" set of a tree and the reduced form of a tree under conditional equations. Actually, the definition of a cycle set and reduced forms of a tree under conditional equations are similar to the ones given in Chapter 3; but for sake of completeness, we give those definitions below.

**Definition 5.6 [Cycle Set of a Tree under CE]**

Let $f$ be in $\Sigma$ and $CE$ be a set of conditional equations associated with $\Sigma$. The set \text{g of } \{g/f \Rightarrow^*_C g, \Rightarrow^+_C f\} is the cycle set of $f$ under $CE$ and is designated by $[f]_{CE}$. The cycle set of $f$ (i.e., $[f]_{CE}$) is said to be "final" if $g_c[f]_{CE}$ and $g \Rightarrow^+_C h$ implies $h \in [f]_{CE}$. The subscript $CE$ will be left out when understood from the context. \qed
Definition 5.7  [Reduced Tree under Conditional Equations]

Let $f \in \Sigma$ and $CE$ be a set of conditional equations. The "reduction of $f$ under $CE$" (i.e., $[f]_{CE}$) is the set

$$[f]_{CE} = \text{NONRED}(f) \cup \{f'/f \in_{CE} f'' \text{ and } f'' \in [f'']_{CE} \text{ and } [f'']_{CE} \text{ is final}\}.$$ 

Furthermore, if $\Theta = \{f_1, \ldots, f_n\}$, $f_i \in \Sigma$ for $i = 1, \ldots, n$, then $[[\Theta]]_{CE} = \bigcup_{i=1}^n [f_i]_{CE}$. The subscript $CE$ will be left out when understood from the context. □

Definition 5.8  [Nonexpanding Conditional Rewrite Rule]

A conditional rewrite rule $<b, <L, R> >$ is "nonexpanding" if $\text{SIZE}(L) \leq \text{SIZE}(R)$. Furthermore, a conditional rewriting system $CE$ is non-expanding if and only if every rule in $CE$ is non-expanding. □

The following lemmas are the obvious extension of Lemmas 3.18, 3.19 to conditional equations. As the proofs are similar, we simply state the lemmas.

Lemma 5.9

Let $CE$ be a set of conditional equations associated with $\Sigma$. Let $P, q \in \Sigma(w, s)$ and $t_i \in \Sigma(\lambda, s_i)$ for $i = 1, \ldots, n$ and $w = s_1 \ldots s_n$. If $[[P]]_{CE} = [[q]]_{CE}$ then

$$[[P]_{t_1, \ldots, t_n}]_{CE} = [[q]_{t_1, \ldots, t_n}]_{CE}.$$ □

Lemma 5.10

Let $CE$ be a set of conditional equations associated with $\Sigma$. Let $P \in \Sigma(w, s)$ and $P' \in \Sigma(w', s)$. Let $t_i \in \Sigma(\lambda, s_i)$ and $t_i' \in \Sigma(\lambda, s_i')$ for
Since conditional rewriting systems (CE) define a binary relation \( \Rightarrow \), thus, all of the concepts of sections 3.5 and 3.6 hold for conditional rewriting systems. This is because of that all of the concepts of sections 3.5 and 3.6 are defined for binary relation \( \Rightarrow \).

5.5. Examples: Power of Conditional Equations

For illustration of the concept of conditional equations, two examples will be specified. As we shall see by a tuple, one of data structure components is a set of "conditional equations", which is associated with signature \( \Sigma \). The first example introduces QUEUE data structure, and the second example will introduce BTREE (binary tree) data structure. Within the second example, we will use the QUEUE data structure for the purpose of binary sort tree.

5.5.1. Example 1: QUEUE

Example 5.11

Let \( N=\{\text{Queue}, \text{Boolean}, \text{Data}\}=\{Q,B,D\} \) and \( \Sigma \) (Figure 5.2) be an N-sorted signature such that

\[ \Sigma(\lambda,D) = \{\text{UNDEFINED},a,b,c,\ldots,z\} \]

\[ \Sigma(\lambda,Q) = \{\text{NEWQ}\} \]

\[ \Sigma(QD,Q) = \{\text{ADDEQ}\} \]

\[ \Sigma(Q,Q) = \{\text{DELETEQ}\} \]

\[ \Sigma(Q,D) = \{\text{FRONTQ}\} \]
\( \Sigma(Q,B) = \{ \text{ISNEWQ} \} \)
\( \Sigma(QQ,Q) = \{ \text{APPENDQ} \} \)
\( \Sigma(\lambda,B) = \{ \text{TRUE, FALSE} \} \)

We assume that QUEUE (signature \( \Sigma \) along with the following equations \( E \) to be defined shortly) is a class of queues intended to be a first-in-first-out storage device whose operations are defined below:

1) NEWQ: returns an instance of the empty queue;
2) ADDQ(q,d): places a new item \( d \) in the queue \( q \) and returns the resulting queue;
3) DELETEQ(q): deletes the oldest item in the queue \( q \) and returns the resulting queue;
4) FRONTQ(q): returns the oldest item in the queue \( q \) leaving the \( q \) unchanged;
5) ISNEW(q): tests if queue \( q \) is empty;
6) APPEND(q_1,q_2): appends \( q_1 \) on left of \( q_2 \) such that the front of \( q_2 \) becomes the front element of the resulting queue and the rear of the resulting queue becomes the rear element of \( q_1 \).

The equations \( (E) \) for QUEUE data structure is given in Figure 5.3. In this example, equations for DELETEQ are used in reduction of any arbitrary sequence of operations to its simplest form without changing its value. An easy way to understand equations is to conceive of the
Figure 5.2 Signature of QUEUE data structure
Figure 5.3 Equations of QUEUE data structure

(1): ISNEWQ[NEWQ] = TRUE
(2): ISNEWQ[ADDQ[X_1X_2]] = FALSE
(3): DELETEQ[NEWQ] = NEWQ
(4): ISNEWQ[X_1] \implies \text{DELETEQ}[ADDQ[X_1X_2]] = NEWQ
(5): \text{not ISNEWQ}[X_1] \implies \text{DELETEQ}[ADDQ[X_1X_2]] = \text{ADDQ} \{ \text{DELETEQ}[X_1]X_2 \}
(6): \text{FRONTQ}[NEWQ] = \text{UNDEFINED}
(7): \text{INSEWQ}[X_1] \implies \text{FRONTQ}[ADDQ[X_1X_2]] = X_2
(8): \text{not ISNEWQ}[X_1] \implies \text{FRONTQ}[ADDQ[X_1X_2]] = \text{FRONTQ}[X_1]
(9): \text{APPENDQ}[X_1, \text{NEWQ}] = X_1
(10): \text{APPENDQ}[X_1, \text{ADDQ}[X_2X_3]] = \text{ADDQ} \{ \text{APPENDQ}[X_1X_2X_3] \}
(11): \text{DELETEQ}[\text{ADDQ}[\text{NEWQ} X_1]] = \text{NEWQ}
set of all queues as being represented by the set of strings consisting of

1) NEWQ as an empty queue; or

2) ADDQ<...ADDQ<ADDQ<NEWQ a₁>a₂>...aₙ> as a queue of length
   n where n≥1 and aᵢ ∈{a,b,...,z} such that the item a₁ is
   at the front and aₙ is at the rear.

The final note is that QUEUE data structure properties cannot be
specified by just simple equations.

5.5.2. Example 2: BINARY TREE

Example 5.12

The concept of a "binary tree" can be used for sorting of
elements. In this example, we consider a binary tree as a set of
nodes which is either empty or a root node and two disjoint binary
trees (called the left and right subtrees). In this example, we
include an operation, SORT, which performs the "inorder traversal"
of a given binary tree. Consider N-sorted signature Σ and equations
E of Example 5.12. Let N' = {Binarytree} u N, Σ' = Σ u Σ₁, and CE = E u C
where N' = {Q,B,D,Binarytree} = {Q,B,D,T} and

Σ₁(λ,T) = {EMPTYTREE,ERROR}
Σ₁(TD,T) = {INSERT,DELETE}
Σ₁(TDT,T) = {MAKE}
Σ₁(T,B) = {ISEMPTYTREE}
Σ₁(T,T) = {LEFT,RIGHT}
\[ \Sigma_1(T,D) = \{ \text{ROOT} \} \]
\[ \Sigma_1(TD,B) = \{ \text{MEMBER} \} \]
\[ \Sigma_1(T,Q) = \{ \text{SORT} \} \]

Let \( CE \) (Figure 5.4) be a set of equations associated with \( \Sigma_1 \). Then signature \( \Sigma' \) along with the equations \( CE \) is called BTREE data structure. The operations for manipulating BTREE data structure is given below.

a. EMPTYTREE: which creates the empty tree;
b. MAKE: which joins two trees together with a new root;
c. ROOT: which accesses the data at the root of a tree;
d. LEFT: which returns the left subtree of a tree;
e. RIGHT: which returns the right subtree of a tree;
f. MEMBER: which tests if a given data item is in the tree;
g. ISEMPTYTREE: which tests if a tree is empty.

In expressing equations only the first two characters of each operation is given and with our specification, an element of BTREE data structure is of the form

\[ <\text{leftsubtree}, \text{root}, \text{rightsubtree}> \]

such that \text{root} is greater than any item in \text{leftsubtree} and less than any item in \text{rightsubtree}. For example, if \( t=\text{MAKE}<t_1 8 t_2> \) where

\[ t_1=\text{INSERT}<\text{INSERT}<\text{INSERT}<\text{EMPTYTREE 5}>9>6> \]
\[ t_2=\text{INSERT}<\text{INSERT}<\text{EMPTYTREE 4}>10> \]

then repetitive application of \( CE \) to \( t \) will yield \( t' \), that is, \( t \xrightarrow{CE} t' \) and using IN instead of INSERT we have:
Figure 5.4 Equations CE of BTREE data structure
and the concept of tree $t'$ may be pictured as Figure 5.5. Applying SORT operation on $t'$ will yield a sorted queue.

5.6. Major Theorem

After defining a few syntactic issues about tree transducers, which are associated with conditional equations we will state a theorem. Likewise, sections 4.4.2. and 4.4.3. we define local consistency, local semiconsistency and global consistency and global semiconsistency under conditional equations.

Definition 5.13 [Conditional Local Consistency and Semiconsistency]

Let $T = (Y_1, Y_2)$ be a tree transducer within $L(Y_1)$ and $L(Y_2)$ and $CE_i$ be a set of conditional equations associated with $\Sigma_i$ for $i=1,2$. Then $T$ is said to be "conditionally locally consistent" if the following conditions (CCL1) and (CCL2) are satisfied.

Also, $T$ is said to be "conditionally locally semiconsistent" from source (target) to target (source) if CCL1 (CCL2) is satisfied.

(CCL1): for all $(b, <L=R>) \in CE_1$,

if \( \text{EVAL}(b_{\Sigma_1} (t_1, ..., t_n)) = \text{TRUE} \), then

if \( L_{\Sigma_1} (t_1, ..., t_n) \in L' \) and

\( R_{\Sigma_1} (t_1, ..., t_n) \in R' \) then

\[ \text{[[} L' \text{]]}_{CE_2} = \text{[[} R' \text{]]}_{CE_2} \]
Figure 5.5 Representation of a binary tree (sort tree)
(CCL2): for all \((b', <L'=R'>) \in CE_2\)

\[
\text{if } L_{\Sigma_2} = t_{i_1}, \ldots, t_{i_n} \text{, and } R_{\Sigma_2} = t_{i_1'}, \ldots, t_{i_n'}, \text{ and } \text{EVAL}(b', (t_{i_1'}, \ldots, t_{i_n'})) = \text{TRUE}, \text{ then } [L]_{CE_1} = [R]_{CE_1} \]

\[\[\]

Definition 5.14 [Conditional Global Consistency and Semiconsistency]

Let \(T = (\gamma_1, \gamma_2)\) be a tree transducer within \(L(\Theta_1)\) and \(L(\Theta_2)\)

and let \(CE_i\) be a set of conditional equations associated with \(\Sigma_i\)

for \(i=1,2\). Then \(T\) is said to be "conditionally globally consistent"

if the following conditions \((CCG1)\) and \((CCG2)\) are satisfied. Also,

\(T\) is said to be "conditionally globally semiconsistent" from source

(target) to target (source) if \(CCG1\) (CCG2) is satisfied.

\[(CCG1): \text{if } [s]_{CE_1} = [t]_{CE_1}, \text{ s} \not\equiv s', \text{ and } t \not\equiv t', \text{ then } [s']_{CE_2} = [t']_{CE_2}.\]

\[(CCG2): \text{if } [s'']_{CE_2} = [t'']_{CE_2}, \text{ s} \not\equiv s'', \text{ and } t \not\equiv t'', \text{ then } [s]_{CE_1} = [t]_{CE_1}.\]

\[\[\]

Definition 5.15 [Conditional Weak Syntactical Honest Tree Transducer]

Let \(T = (\gamma_1, \gamma_2)\) be a tree transducer such that \(\Sigma_i = (\Sigma_i', \Sigma_i, \eta_i, \pi_i)\)

for \(i=1,2\) and \(\text{TRANS}(T,a) = ((h_1(w), h_2(w))/h_i): T_{\Sigma_i'} \rightarrow T_{\Sigma_i}\) and \(h_i\) is a
unique homomorphism for i=1,2). Let CE₁, CE₂ be sets of conditional equations for source and target, respectively. Tree transducer T is "conditional weak syntactical honest" if

i) T is conditionally locally semiconistent from source to target;

ii) CE₁ and CE₂ are nonexpanding;

iii) T is mono from source to target;

iv) SBASE={h₁(w)/|w|<2}, TBASE={h₂(w)/|w|<2}, if SB is the least element of L(SBASE), b₁, b₂ ∈ SB, b₁ = b₂, c₁, c₂ ∈ CE₁, d₁, d₂ ∈ CE₂ for i=1,2 implies that there exists a TB in the least element of L(TBASE) and d₁, d₂ ∈ TB;

v) [\[ π₁(P) \]\] CE₁ = [\[ π₁(q) \]\] CE₁ implies [\[ π₂(P) \]\] CE₂ = [\[ π₂(q) \]\] CE₂

where P, q ∈ Σ'.

Theorem 5.16

If T is a conditionally weak syntactic honest tree transducer, then T is a weak SYNIMPL tree transducer.

Proof: The proof of this theorem is similar to the proof of Theorem 4.16 and will not be repeated here.

5.7. Summary and Results

We have extended the concepts of Chapter 3 to conditional equations. In the next chapter, we will see the application of such equations to define data structures. Simple equations are not able to specify the properties of complex data structures. This draw-
back is removed by introducing conditional equations.

We have also examined the proof of constructing an algebraic tree transducer whose properties are related to conditional equations.

We believe that in many ways, the benefits of using conditional equations to simple equations are similar to the benefits of using "context-free languages" to "regular languages".
6. DATA STRUCTURE IMPLEMENTATION AND CORRECTNESS

In this chapter, we develop a general technique to implement one data structure (the source data structure) in terms of another (the target data structure). In the algebraic framework, we will consider a data structure as a signature $\Sigma$ along with a set of equations. Given algebraic specifications of data structures $d_1$ and $d_2$, an "implementation of $d_1$ by $d_2$" is defined on the syntactical level of specifications and on the semantic level of algebras.

First, we give the basic definitions and constructions for specification of data structures. Secondly, implementation of the source data structure by the target data structure will be defined. Then, correctness of implementation will be considered. It will be shown that the concept of consistent and semiconsistent tree transducers are useful for provably correct implementations. That is, the correctness of an implementation can be demonstrated from concepts of consistency.

Finally, we will show that there exist data structures which have finite conditional specifications but only infinite equational specifications. Implicitly, this chapter shows that initial $\Sigma$-algebras are important algebras because they are not only can be used for defining the semantics but also used for proving properties of data structures.
6.1. Related Work and Discussion

The algebraic approach to the specification, correctness and implementation of data structures has been examined by several researchers, including Goguen, Thatcher, Wagner, and Wright [10], Guttag [13], and Goguen [9] to mention a few. Guttag [13] and Goguen [9] have used algebraic equations for an implementation of a data structure. Related work has been done by Hoare [17]. In our approach, the syntax associated with a data structure is merely the signature of a context-free grammar and its semantics is given by an algebra.

Guttag's technique is programming language independent. Goguen's work implements a data structure by using algebraic homomorphisms. Parsian [36], similar to Guttag, developed the following idea. Suppose a data structure $d$ along with its operations is to be implemented in a language $L_2$. Let $L_1$ be another language that includes all primitives for manipulating the data structure $d$. If we can now translate from $L_1$ to $L_2$, we would have "implemented $d$ in $L_2"."

Thus, there are several approaches to make the notion of implementation of a data structure precise. Our method is one way we can implement a source data structure by a target data structure and prove the implementation correct. We introduce a conceptually simple but adequate notion of data structure implementation in terms of tree transducers and equations. Our approach defines the set of rules (a tree transducer) to implement a new (source) data structure by an old (target) data structure.
Guttag [13] argues that implementation of a data structure \( d \) is a mapping from \( d \) to other data structures. For example, a stack data structure can be implemented by a pair comprising an array and a pointer, where the pointer indicates the position of top element of the stack in the array.

Liskov et al. [33] state that "the behavior of the data objects (data structure) is expressed most naturally in terms of a set of operations that are meaningful for those objects and this set includes operations to create objects (data structure elements), to obtain information from them, and possibly to modify them".

In our approach to implement a source data structure by target data structure, we use the following strategy. If each operation and compound operation of a source data structure \( (d_1) \) can be simulated (expressed in a tree transducer) by operations synthesized from those in target data structure \( (d_2) \), then \( d_1 \) is said to have been implemented by \( d_2 \).

In this approach we give a precise definition for correctness of implementation of data structures. Correctness of an implementation can be examined from two viewpoints: syntax and semantics. We will consider both of these views.

Finally, recent attention in implementation of data structures has resulted in a number of data abstraction languages. For example, ADA [44] and CLU [33] are programming languages designed to support the use of abstractions in program construction.
6.2. Algebraic Specification of a Data Structure

Algebraic specification techniques have been examined by several researchers, including Liskov and Zilles [34], Goguen [8, 9], and Guttag [13, 14]. The intuitive idea behind algebraic specification of data structures is that data are defined by the effects of operations (functions) on it, rather than by some particular representation.

Basically, a data structure is a set of operations (Σ) whose behavior is to be governed by a set of equations (E). We now turn to a formal definition of a specification of a data structure.

6.2.1. Definition of a data structure

Definition 6.1 [Specification of a Data Structure]

Let $G$ be a context-free grammar, $D=(G,A)$ be an LDS, and $Σ$ be the signature of $G$. Let $E$ be a set of equations associated with $Σ$ that satisfies $A$. Then $d=(Σ,A,E)$ is a specification of a data structure, called $d$. Furthermore,

i) if $E=∅$ then $d=(Σ,A,E)$ is called a "free data structure" (FDS);

ii) if $E$ is a set of simple equations, then $d=(Σ,A,E)$ is called a "simple data structure" (SDS);

iii) if $E$ is a set of conditional equations, then $d=(Σ,A,E)$ is called a "complex data structure" (CDS).

We note that if $d=(Σ,A,∅)$, then every element of data structure $d$ is in irreducible form and cannot be reduced further. In order to illustrate the definition of a data structure, consider the following example.
6.2.2. Example of a data structure

Example 6.2

Let \( N = \{\text{Array}, \text{Domain type}, \text{Range type}\} = \{A, D, R\} \) and \( \Sigma \) be an \( N \)-sorted signature such that

\[
\begin{align*}
\Sigma(\lambda, A) &= \{\text{NEWARRAY}\} \\
\Sigma(\text{ADR}, A) &= \{\text{ASSIGN}\} \\
\Sigma(\text{AD}, R) &= \{\text{ACCESS}\} \\
\Sigma(\lambda, R) &= \{\text{UNDEFINED}\}
\end{align*}
\]

Let \( B \) be a \( \Sigma \)-algebra\(<B_A, B_D, B_R>, \{\text{NEWARRAY}_B, \text{ASSIGN}_B, \text{ACCESS}_B, \text{UNDEFINED}_B\}\>\) informally defined as:

1) NEWARRAY is an empty array;
2) ASSIGN\((a, i, v)\) means that the array identical to \( a \) except possibly in the \( i \)-th position where the value is \( v \);
3) ACCESS\((a, i)\) returns the value in position \( i \) of the array \( a \); if \( a \) is a NEWARRAY then it returns UNDEFINED.

These properties can be expressed by the following equations:

\[
\begin{align*}
(c1) \text{ACCESS}[\text{NEWARRAY } x_1] &= \text{UNDEFINED} \\
(c2) (x_2=x_4) &\Rightarrow \text{ACCESS}[\text{ASSIGN}[x_1 x_2 x_3 x_4]] = x_3 \\
(c3) \text{not } (x_2=x_4) &\Rightarrow \text{ACCESS}[\text{ASSIGN}[x_1 x_2 x_3 x_4]] = \text{ACCESS}[x_1 x_4]
\end{align*}
\]

Let \( E = \{c1, c2, c3\} \), then ARRAY\(=\langle \Sigma, B, E \rangle \) is complex data structure because \( E \) is a set of conditional equations.

The preceding example illustrates the need to separate the specification and implementation of a data structure. That is, in the specification phase we are not stating how the operations will be implemented, but merely specifying the properties of those
operations. That is, the specification of a data structure must be its abstraction; this should be all that a user of the data structure needs to know about it. The implementation of a data structure is a separate issue and may be done by a different person. That is, the implementation should be hidden from the user.

6.3. Equivalence of Data Structures

This section defines when data structures \( \mathbf{d}_1 \) and \( \mathbf{d}_2 \) are "equivalent". We then state and prove a theorem, which indicates that complex data structures are "more powerful" than simple data structures. Given \( \mathbf{d}_1 = (\Sigma, A, E_1) \) and \( \mathbf{d}_2 = (\Sigma, A, E_2) \), the question is "are \( \mathbf{d}_1 \) and \( \mathbf{d}_2 \) equivalent"? A positive answer to this question is useful, because if \( \mathbf{d}_1(\mathbf{d}_2) \) is known to be correct then their equivalence will yield the correctness of \( \mathbf{d}_2(\mathbf{d}_1) \). Besides if \( \mathbf{d}_1 \) and \( \mathbf{d}_2 \) are equivalent and \( \mathbf{d}_2 \) has an efficient implementation, then we can choose \( \mathbf{d}_2 \).

First, a method for checking the equivalence of data structures will be presented. Then, the method will be illustrated by the example of linear lists.

6.3.1. Method for checking equivalent data structures

In order to define equivalence of data structures, we need to define the concept of a "theorem in \( E \)."

**Definition 6.3 [Theorem in \( E \)]**

Let \( E \) be a set of equations associated with \( E \). Let \( f_1, f_2 \) be in \( E \). Then \( f_1 \equiv f_2 \) is a "theorem in \( E \)" if and only if \( f_1 \preceq f \) and \( f_2 \preceq f \) for some \( f \) in \( \bar{E} \).
Let \( d_1 = (\Sigma, A, E_1) \) and \( d_2 = (\Sigma, A, E_2) \) be data structures. The method for checking equivalence consists of verifying that each rewrite rule of \( E_1 \) is a theorem of \( E_2 \), and vice versa. The following is a formal definition of equivalent data structures.

**Definition 6.4** [Equivalence of Simple Data Structures]

Let \( d_1 = (\Sigma, A, E_1) \) and \( d_2 = (\Sigma, A, E_2) \) be simple data structures. If for every \( (L = R) \in E_1 \), \( L = R \) is a theorem of \( E_2 \) and for every \( (L' = R') \in E_2 \), \( L' = R' \) is a theorem of \( E_1 \), then \( d_1 \) and \( d_2 \) are "equivalent" data structures.

**Definition 6.5** [Equivalence of Complex Data Structures]

Let \( d_1 = (\Sigma, A, C_1) \) and \( d_2 = (\Sigma, A, C_2) \) be complex data structures and \( \nu \) be a semantic homomorphism from \( T_\Sigma \) to \( A \). If for every \( (b, <L = R>) \in C_1 \), and if \( \text{EVAL}(b_A(y_1, \ldots, y_n)) = \text{TRUE} \) and \( y_i = \nu(t_i) \) for \( i = 1, \ldots, n \), implies that \( L_\Sigma(f) = R_\Sigma(f) \) is a theorem of \( C_2 \) for \( f = (f_1, \ldots, f_n) \); and if for every \( (b', <L' = R'>) \in C_2 \), and if \( \text{EVAL}(b'_A(w_1, \ldots, w_m)) = \text{TRUE} \) and \( w_i = \nu(t_i) \) for \( i = 1, \ldots, m \), implies that \( L'_\Sigma(t) = R'_\Sigma(t) \) is a theorem of \( C_1 \) for \( t = (t_1, \ldots, t_m) \), then \( d_1 \) and \( d_2 \) are equivalent data structures.

6.3.2 Example of equivalent data structures

In order to illustrate Definition 6.4 consider the following example.

**Example 6.6**

Let \( N = \{L, D\} \) and \( \Sigma \) be an \( N \)-sorted signature such that

\[ \Sigma(\lambda, L) = \{\text{NIL}\} \]
Furthermore, let $A$ be a $\Sigma$-algebra and $E_1, E_2$ be a set of equations (Figure 6.1) associated with $\Sigma$. Our claim is that $d1=(\Sigma, A, E_1)$ and $d2=(\Sigma, A, E_2)$ are equivalent data structures. In order to prove this, we first check that for each equation $(L, R) \in E_2$ we can derive $L=R$ in $E_1$. For instance consider

$$c1=\langle \text{MAKE}[X_1], \text{CONS}[X_1 \text{ NIL}] \rangle$$

Then

$$\text{CONS}[X_1 \text{ NIL}] \Rightarrow \text{APPEND}[\text{MAKE}[X_1 \text{ NIL}]]$$

$$\Rightarrow \text{MAKE}[X_1]$$

Thus, $c1$ is a theorem of $E_1$. Verifying the other equations are similar and will not be repeated here.

6.3.3. Power of complex data structures

Having a method for checking equivalence of data structures, we can show that complex data structures are "more powerful" than simple data structures.

**Theorem 6.7**

There exists a complex data structure with a finite conditional specification, which is not equivalent to any simple data structure, with a finite number of simple equations.

**Proof:** Consider the complex data structure $\text{ODDEVEN}=(\Sigma, A, E)$. 
\( E_1: \)

(e1): \( \text{APPEND}(\text{NIL } X_1) = X_1 \)

(e2): \( \text{APPEND}(\text{MAKE}(X_1)\text{NIL}) = \text{MAKE}(X_1) \)

(e3): \( \text{APPEND}(\text{APPEND}(\text{MAKE}(X_1)X_2)X_3) = \text{APPEND}(\text{MAKE}(X_1)\text{APPEND}(X_2X_3)) \)

(e4): \( \text{CONS}(X_1X_2) = \text{APPEND}(\text{MAKE}(X_1)X_2) \)

\( E_2: \)

(c1): \( \text{MAKE}(X_1) = \text{CONS}(X_1 \text{ NIL}) \)

(c2): \( \text{APPEND}(\text{NIL } X_1) = X_1 \)

(c3): \( \text{APPEND}(\text{CONS}(X_1X_2)X_3) = \text{CONS}(X_1 \text{APPEND}(X_2X_3)) \)

Figure 6.1 Equations \( E_1, E_2 \)
Let Σ be an N-sorted signature, A be a Σ-algebra, and E be a set of conditional equations for ODDEVEN where

\[ N = \{\text{Even, Natural number}\} = \{V, R\} \]

\[ \Sigma(\lambda, R) = \{\text{ZERO}\}, \Sigma(\lambda, V) = \{\text{ODD}\} \]

\[ \Sigma(R, R) = \{\text{SUCC}\}, \Sigma(R, V) = \{\text{RED}\} \]

A = \langle A, \Phi \rangle where A = \{A_V, A_R\} with

\[ A_R = \{0, 1, 2, 3, 4, \ldots\} \]

\[ A_V = \{\text{ODD}, 0, 2, 4, \ldots\} \]

\[ F = \{\text{ZERO}_A, \text{SUCC}_A, \text{ODD}_A, \text{RED}_A\} \]

\[ \text{ZERO}_A = 0; \text{SUCC}_A(x) = x + 1; \text{ODD}_A \]

\[ \text{RED}_A(x) = \begin{cases} x & \text{if } x \text{ is an even number} \\ \text{ODD otherwise} & \end{cases} \]

The properties of the above functions can be expressed by the following equations, where E = \{e_1, e_2\}.

(e1): \text{RED}[\text{SUCC}[\text{ZERO}]] = \text{ODD}

(e2): (\text{RED}[x] = \text{ODD}) \Rightarrow \text{RED}[\text{SUCC}[\text{SUCC}[x]]] = \text{ODD}

Now suppose d' = (Σ, A, E') is a simple data structure and E' = \{e_1', \ldots, e_k'\} for some finite k. Suppose d' is equivalent to ODDEVEN, we will now derive a contradiction. Equation e_1 in E is a simple equation and could be part of E'. Obviously e_1 could be a theorem of E'. Considering equation e_2, we are now going to find for what values of x \text{RED}[\text{SUCC}[\text{SUCC}[x]]] = \text{ODD} is a theorem of E'.

Define \text{SUCC}^n[x] for n \geq 0 as follows:

1. \text{SUCC}^0[x] is x

2. \text{SUCC}^n[x] is \text{SUCC}[\text{SUCC}^{n-1}[x]].
Now, \( \text{EVAL}(\text{RED}_A[W] = \text{ODD}_A) \) is TRUE if and only if \( W = \text{SUCC}_A^{2n+1}([\text{ZERO}] \) for \( n > 0 \). Hence, if the consequent of \( e_2 \),

\[ \text{RED}([\text{SUCC}[\text{SUCC}[X]]]) = \text{ODD}, \]

is to be a theorem of \( E' \) we must have \( W = \mu(X) \) and hence \( X = \text{SUCC}_A^{2n+1}([\text{ZERO}] \). Thus, we have to show that

\[ \text{RED}([\text{SUCC}[\text{SUCC}^{2n+1}[\text{TWO}]]]) = \text{ODD} \]

is a theorem of \( E' \) for \( n > 0 \).

We now claim that \( \text{RED}([\text{SUCC}[\text{SUCC}^{2n+1}[\text{TWO}]]]) \) and \( \text{ODD} \) are irreducible forms in \( E' \). This is proved as below:

Suppose there are \( k \) equations for \( E' \). That is, \( E' = \{e_1, \ldots, e_k\} \).

Find an \( m \) such that none of \( e_1, \ldots, e_k \) have term \( \text{L} \) (left-hand side) or \( \text{R} \) (right-hand side) equal to

\[ \text{RED}([\text{SUCC}^{2m-1}[X]]) \]

or \[ \text{RED}([\text{SUCC}^{2m}[X]]) \]

Now, let

\[ L_E = \{L/\langle L, R \rangle \in E'\}, \quad \text{and} \]

\[ R_E = \{R/\langle L, R \rangle \in E'\} \]

\( L_E \) and \( R_E \) are finite sets. Thus, there exists \( m \) such that

\[ \text{RED}([\text{SUCC}^{2m-1}[X]]) \notin L_E \cup R_E; \]

and \[ \text{RED}([\text{SUCC}^{2m}[X]]) \notin L_E \cup R_E. \]

We will show that it is impossible to have as theorems of \( E' \)

\[ \text{RED}([\text{SUCC}^{2m-1}[X]]) = \text{ODD}; \]

\[ \text{and RED}([\text{SUCC}^{2m}[X]]) = X \] \hspace{1cm} (**1)
or

\[ \text{RED}[\text{SUCC}^{2m-1}[X]] = X; \]
\[ \text{and RED}[\text{SUCC}^{2m}[X]] = \text{ODD} \quad (*2) \]

Without loss of generality let us suppose that (*1) is true. We will derive a contradiction. The contradiction for (*2) will be similar. Suppose (*1) is true. Suppose \( X = \text{SUCC}^P[\text{ZERO}] \), then the following are theorems of \( E' \)

\[ \text{RED}[\text{SUCC}^{2m+P-1}[\text{ZERO}]] = \text{ODD}; \]
\[ \text{and RED}[\text{SUCC}^{2m+P}[\text{ZERO}]] = \text{SUCC}^P[\text{ZERO}] \]

But this is not true because if \( P \) is odd, then

\[ \text{RED}[\text{SUCC}^{2m+P-1}[\text{ZERO}]] = \text{ODD} \]

is no longer a theorem of \( E' \), on the other hand if \( P \) is even, then

\[ \text{RED}[\text{SUCC}^{2m+P}[\text{ZERO}]] = \text{SUCC}^P[\text{ZERO}] \]

cannot be a theorem of \( E' \), for if it were, then \( \text{RED}_A(2m+P) = \text{RED}_A(P) \) where \( P \) is even, which is impossible by definition of \( \text{RED}_A \). This is the end of our claim. Thus, neither (*1) nor (*2) are theorems in \( E' \). But we know that either (*1) or (*2) must be theorem of \( E \). Thus,

\[ \text{RED}[\text{SUCC}[\text{SUCC}[\text{SUCC}^{2n+1}[\text{ZERO}]]]] = \text{ODD} \]

must be in \( E' \) for \( n \geq 0 \). Hence, \( E' \) must have an infinite number of equations, and this is in contradiction with our assumption that \( E' \) has a finite number of equations.

\[ \square \]

Hence, the preceding theorems shows that conditional equations are "more powerful" than simple equational ones, in the sense that
there exist finite conditional specifications which require infinite equational specifications.

At the end of this section, we give an example of how a complex data structure can be equivalent with a simple data structure. One purpose of this example is to show that conditional equations implicitly contain the "if-then-else" operation, which has been used in the Guttag's work [14, 16]; and the other purpose is to illustrate Definition 6.5.

Example 6.8

Let $\Sigma$ be an $\{V,R\}$ sorted signature, $A$ be a $\Sigma$-algebra, $E$ be a set of conditional equations (Figure 6.2), and $E'$ be a set of simple equations (Figure 6.2) such that

$\Sigma(\lambda,R) = \{\text{ZERO}\}$, $\Sigma(R,R) = \{\text{SUCC, EVEN}\}$

$\Sigma(\lambda,V) = \{\text{ODD}\}$, $\Sigma(RV,V) = \{\text{IF}\}$

$\Sigma(R,V) = \{\text{RED}\}$

$A = \langle A,F \rangle$ where $A = \{A_V,A_R\}$ with

$A_R = \{0,1,2,3,4,\ldots\}$

$A_V = \{\text{ODD},0,2,4,\ldots\}$

$F = \{\text{ZERO}_A,\text{SUCC}_A,\text{ODD}_A,\text{RED}_A,\text{EVEN}_A,\text{IF}_A\}$ with

$\text{ZERO}_A = 0$; $\text{SUCC}_A(x) = x+1$; $\text{ODD}_A = \text{ODD}$; $\text{RED}_A(x) = \{\text{if } x \text{ is an even number otherwise}\}$

$\text{IF}_A(x,t_1,t_2) = \begin{cases} t_1 & \text{if } x = 0 \\ t_2 & \text{otherwise} \end{cases}$

$\text{EVEN}_A(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{if } x \text{ is odd} \end{cases}$

The properties of the above functions have been expressed in Figure 6.2.
Conditional equations E:

(1): (X = ZERO) or (X = SUCC(ZERO)) ⇒ EVEN[X] = X

(2): (EVEN[X] = SUCC(ZERO)) ⇒ RED[X] = ODD

(3): EVEN[SUCC[SUCC[X]]] = EVEN [X]

(4): (X₁ = ZERO) ⇒ IF[X₁X₂X₃] = X₂

(5): not (X₁ = ZERO) ⇒ IF[X₁X₂X₃] = X₃

Simple equations E':

(6): EVEN[ZERO] = ZERO

(7): EVEN[SUCC(ZERO)] = SUCC[ZERO]

(8): EVEN[SUCC[SUCC[X]]] = EVEN[X]

(9): IFZERO X₁X₂] = X₁

(10): IF[SUCC[X₁]X₂ X₃] = X₃


Figure 6.2 Conditional and simple equations
Let $d=(\Sigma,A,E)$ and $d'=(\Sigma,A,E')$ be complex and simple data structures. We now will show that $d$ and $d'$ are equivalent. First, we prove that the consequence of equation (1) is a theorem of $E'$. Thus, we have to prove that whenever $X=\text{ZERO}$ or $X=\text{SUCC[ZERO]}$ then $\text{EVEN}[X]=X$ is a theorem of $E'$.

Let $X=\text{ZERO}$, then $\text{EVEN}[\text{ZERO}]=\text{ZERO}$; since $\text{EVEN}[\text{ZERO}]=\text{ZERO}$ is an equation of $E'$ thus $\text{EVEN}[\text{ZERO}]=\text{ZERO}$ is a theorem of $E'$. Similarly, $\text{EVEN}[\text{SUCC[ZERO]}]=\text{SUCC[ZERO]}$ is a theorem of $E'$.

It can be verified that equations (2), (3), (4), (5) are theorems of $E'$ and also, it can be proved that equations (6), (7), (8), (9), (10), and (11) are theorems of $E$. Hence $d$ and $d'$ are equivalent. 

6.4. Implementation of a Data Structure

Implementation of data structures will be defined on syntactical and semantical levels. In our work, we use algebraic equations and tree transducers for the implementation of data structures.

Given data structures $d_1=(\Sigma_1,A_1,E_1)$ and $d_2=(\Sigma_2,A_2,E_2)$, tree transducer $T$ is a syntactical implementation of $d_1$ by $d_2$ if $(t_1,t'_1) \in \text{TRANS}(T,s)$ for $i=1,2$ and $[[t_1]]_{E_1}=[[t'_1]]_{E_1}$ implies $[[t_1]]_{E_2}=[[t'_1]]_{E_2}$. Also, $T$ becomes semantical implementation if $T$ is a syntactical implementation and if $(t_1,t_2) \in \text{TRANS}(T,s)$ implies that $t_1$ and $t_2$ are semantically equivalent.

Before introducing the implementation of data structures, we define the following concepts.
Definition 6.9 [Syntactically Equivalent, Semantically Equivalent]

Let $d_1=(\Sigma_1, A_1, E_1)$ and $d_2=(\Sigma_2, A_2, E_2)$ be data structures. Let

$\tau=(\gamma_1, \gamma_2)$ be a tree transducer such that $\gamma_i=(\Sigma_i, E_i, \eta_i, \pi_i)$ and $\mu_i: T_{\Sigma_i} \rightarrow A_i$ for $i=1,2$. If $(t_1, t_2) \in \text{TRANS}(\tau, s)$ and $\mu_1(t_1) = \mu_2(t_2)$ then we say that $t_1$ and $t_2$ are semantically equivalent. If $[[t_1]]_{E_1} = [[t_2]]_{E_1}$ and $(t_i, t'_i) \in \text{TRANS}(\tau, s)$ for $i=1,2$ implies $[[t'_1]]_{E_2} = [[t'_2]]_{E_2}$, then

1) $t_1$ and $t_2$ are syntactically equivalent within $E_1$.
2) $t'_1$ and $t'_2$ are syntactically equivalent within $E_2$.
3) $t_1, t'_1$ and $t_2, t'_2$ are syntactically equivalent within $E_1$ and $E_2$. •

6.4.1. Syntactical level of implementations

In this section we will consider syntactical level of implementation of data structures. Let $d_1=(\Sigma_1, A_1, E_1)$ and $d_2=(\Sigma_2, A_2, E_2)$. Also, let $\tau$ be a tree transducer which converts elements of $d_1$ into elements of $d_2$. If $[[t_1]]_{E_1} = [[t_2]]_{E_1}$ and $t_1 = t'_1$ and $t_2 = t'_2$ implies $[[t'_1]]_{E_2} = [[t'_2]]_{E_2}$ then $d_1$ is said to have been "implemented syntactically" by $d_2$, via tree transducer $\tau$, then it does not necessarily mean that $(t_1, t_2) \in \text{TRANS}(\tau, s)$ implies $t_1$ and $t_2$ are semantically equivalent.

Intuitively, if $d_1$ has been "implemented syntactically" by $d_2$ via tree transducer $\tau$, then $\tau$ takes elements of an equivalence class from $d_1$ and associates these elements with an equivalence class in $d_2$. 
That is, if $A=\{t/[[t']] E_1=[[t]] E_1$ for some $t'$ over $\Sigma_1^t$ then if $t' \in t''$ and $B=\{y/x^t y$ and $x \in A\}$ then $[[t'']] E_2=[[y]] E_2$ for every $y \in B$.

We will prove that if $T$ is locally consistent, then it is possible to correctly map an equivalent class of elements of data structure $d_1$ into an equivalent class of elements of $d_2$. The following definition is the generalization of the Definition 4.15.

**Definition 6.10** [Syntactical Data Structure Implementation]

Let $D_i=(G_i,A_i)$ be an LDS, $\Sigma_i$ be the signature of $G_i$, and $\nu_i: \Sigma_i \rightarrow A_i$ be the semantic homorphism of $D_i$ for $i=1,2$. Let $d_i=(\Sigma_i,A_i,E_i)$ and $d_2=(\Sigma_2,A_2,E_2)$ be source and target data structures respectively. Then $d_1$ is said to have been "implemented syntactically" in $d_2$ by a tree transducer $T=(Y_1,Y_2)$ where:

1) $Y_i=(\Sigma_i,\nu_i,\eta_i,\pi_i)$ for $i=1,2$; and
2) for $t_1,t_2 \in T_{\Sigma_1}$ and $t_1 \in t_1', t_2 \in t_2'$ and $[[t_1]] E_1=[[t_2]] E_1$ implies $[[t_1']] E_2=[[t_2']] E_2$.

Furthermore $T$ is called a "syntactical implementation" (SYNIMPL) tree transducer.

Thus, syntactical implementation of $d_1$ in $d_2$ implies that we are only interested in converting the elements of source data structure ($d_1$) into target data structure ($d_2$) without checking that those elements have the same meaning. Basically, syntactical implementation of $d_1$ in $d_2$ means that if $[[t_1]] E_1=[[t_2]] E_2$ and $t_1 \in t_1'$ and $t_2 \in t_2'$ then $[[t_1']] E_2=[[t_2']] E_2$. After defining the concept of a "tree
"translation", we will introduce the formal definition of semantical data structure implementations.

### 6.4.2. Semantical Level of Implementations

**Definition 6.11 [Tree Translation]**

Let $D_i = (G_i, A_i)$ be an LDS and $E_i$ be the signature of $G_i$ for $i = 1, 2$. Let $T = (\gamma_1, \gamma_2)$ be a tree transducer such that $\gamma_i = (\Sigma_i', \Sigma_i, \eta_i, \pi_i)$ for $i = 1, 2$, and $\text{TRANS}(T, s)$ be a family of tree transductions such that $\text{TRANS}(T, s) = \{(t_1, t_2) / t_i = h_i(t) \}$ for $i = 1, 2$ and $t \in T_{E_i}$, where $h_i$ is a unique homomorphism from $T_{E_i}$ to $T_{\Sigma_i}$. Let $\mu_i : T_{E_i} \to A_i$ be the semantic homomorphism of $D_i$. Then a tree transduction $\text{TRANS}(T, s)$ is a "tree translation" if $\mu_1(t_1) = \mu_2(t_2)$ for all $(t_1, t_2) \in \text{TRANS}(T, s)$.

**Definition 6.12 [Semantical Data Structure Implementation]**

Let $D_i = (G_i, A_i)$ be an LDS, $E_i$ be the signature of $G_i$ and $\mu_i : T_{E_i} \to A_i$ be the semantic homomorphism of $D_i$ for $i = 1, 2$. Let $d_1 = (\Sigma_1, A_1, E_1)$ and $d_2 = (\Sigma_2, A_2, E_2)$ be source and target data structures respectively. Then $d_1$ is said to have been "implemented semantically" by $d_2$ if

1) $d_1$ has been implemented syntactically by $d_2$ via tree transducer $T$; and

2) $\text{TRANS}(T, s)$ is a tree translation for every $s$ in $N'$, where $N'$ is the sort of common representable signature of $T$.

Also, $T$ will be called a semantical implementation (SEMIMPL) tree transducer.
Semantical data structure implementation of d1 by d2 will imply that if t1 is an element of d1 and t2 is an element of d2, (where T is a SEMIMPL tree transducer) then t1 and t2 are semantically equivalent.

6.5. Correctness of Implementation of Data Structures

Much work on proof techniques for proving correctness of implementation of data structures has been done using Hoare-like [17, 18] axioms. Goguen, Thatcher, and Wagner [11] have discussed techniques of "canonical term algebras" for correctness proofs; [11] also gives a definition of "implementation" within the initial algebra framework. In [11], they state that the whole idea of specifications is to be able to use them to check the correctness of implementations.

Actually, correctness problems arise in two separate but related directions: one wants to be sure one's specification is correct; as well as to be sure that one's implementation matches the specification. We will assume that the specification is correct. That is, we will assume that properties of the operation of a data structure have been correctly specified. We will be concerned only with the correctness of an implementation.

Let d1=(S1,A1,E1) and d2=(S2,A2,E2) be source and target data structures, and T be a SYNIMPL tree transducer. Then, we say that d1 has "correctly been implemented syntactically" by d2 if T is a globally consistent tree transducer. Furthermore, we say that d1 has "correctly
been implemented" by $d_2$ if $T$ is not only a SYNIMPL tree transducer but also $\text{TRANS}(T,s)$ is a tree translation for every possible $s$.

**Definition 6.13**

Let $d_1=(\Sigma_1,A_1,E_1)$ be a source data structure and $d_2=(\Sigma_2,A_2,E_2)$ be a target data structure and $T=(\gamma_1,\gamma_2)$ be a SYNIMPL tree transducer. Then

(i) implementation of $d_1(d_2)$ with respect to $d_2(d_1)$ is "syntactically correct" if $T$ is "globally semiconsistent" from source (target) to target (source).

(ii) implementation of $d_1(d_2)$ with respect to $d_2(d_1)$ is "semantically correct" if

(a) $T$ is globally consistent; and

(b) $\text{TRANS}(T,s)$ is a tree translation.

6.6. Development of Correct Implementations

In this section, we first investigate sufficient conditions to be imposed on tree transducers and equations to guarantee that they have induced correct implementations. Then, we can use these conditions for establishing correct implementation of data structures. The least elements of the lattices developed in Chapter 3 will be found useful.

6.6.1. Development of SYNIMPL tree transducer

Let $d_1=(\Sigma_1,A_1,E_1)$ and $d_2=(\Sigma_2,A_2,E_2)$ be source and target data structures. Further, let $T=(\gamma_1,\gamma_2)$ be a tree transducer such that
\( \gamma_i = (\Sigma_i, \Xi_i, \eta_i, \pi_i) \) for \( i = 1, 2 \). First, we will impose sufficient conditions on \( E_1, E_2, \) and \( T \). Then we will prove that if those conditions are satisfied, then \( T \) is a SYNIMPL tree transducer.

The following definition is the generalization of the Definition 4.14.

**Definition 6.14 [Syntactical Honesty]**

Let \( d_1 = (\Sigma_1, A_1, E_1) \) and \( d_2 = (\Sigma_2, A_2, E_2) \) be source and target data structures. Let \( T = (\gamma_1, \gamma_2) \) be a tree transducer such that \( \text{TRANS}(T, a) = \{(h_1(w), h_2(w))/h_i: \Sigma_i \rightarrow T \rightarrow \Sigma_i, \ h_i \text{ is a unique homomorphism for } i = 1, 2\} \).

Then \( T \) is "syntactically honest" if

i) \( T \) is locally semiconsistent from source to target;

ii) \( E_1 \) and \( E_2 \) are non-expanding;

iii) \( T \) is "mono"; that is, if \( t_i = h(w_i) \) for \( i = 1, 2 \) and \( t_1 \overset{\star}{\rightarrow} t_2 \) then \( |w_1| \geq |w_2| \).

iv) Let \( S\text{BASE} = \{h_1(w)/|w| \leq 2\} \), \( T\text{BASE} = \{h_2(w)/|w| \leq 2\} \); If \( SB \) the least element of \( L(S\text{BASE}) \), \( b_1, b_2 \in SB \), \( b_i \overset{E_1}{\rightarrow} c_i \), \( C_i \overset{T}{\rightarrow} d_i \) for \( i = 1, 2 \) implies that there exists a \( TB \) in the least element of \( L(T\text{BASE}) \) and \( d_1, d_2 \in TB \);

v) Let \( S\text{OURCE} = \{\pi_1(P)/P \in \Sigma'\} \), \( T\text{ARGET} = \{\pi_2(P)/P \in \Sigma'\} \); \( S \) the least element of \( L(S\text{OURCE}) \), \( T \) the least element of \( L(T\text{ARGET}) \), \( a, c \in S \), \( b \in T \), and \( (a, b) \in T \) implies that \( (c, d) \in T \) if and only if \( d \in T \).
vi) $h_1(w)=t$, $|w|>2$, then there exists a $v$ with $h_1(v)=t'$, $|v|>|w|$, and $t' \frac{t_1}{E_1} t$.

Note that Definition 6.14 for a syntactically honest tree transducer imposes some additional constraints from the definition of the weak syntactically honest tree transducer (Definition 4.14). In particular, clauses (v), (vi) of definition 6.14 are needed to prove the generalization of Theorem 4.17 that we are aiming at.

### 6.6.2. Correctness of SYNIMPL tree transducer

The following theorem establishes that every syntactically honest tree transducer within $d_1$ and $d_2$ gives rise to a SYNIMPL tree transducer. Thus, the process of constructing a syntactical implementation can often be reduced to constructing syntactically honest tree transducer. The following theorem is the generalization of the theorems presented in Chapter 4. That is, in Chapter 4, we proved that if $A \frac{t_1}{E_1} B$, $A \frac{t_2}{E_2} C$, $B \frac{t_3}{E_3} D$, then $[[C]]_{E_2} = [[D]]_{E_2}$. Here, we prove that if $[[A]]_{E_1} = [[B]]_{E_1}$, $A \frac{t_2}{E_2} C$, $B \frac{t_3}{E_3} D$, then $[[C]]_{E_2} = [[D]]_{E_2}$.

**Theorem 6.15**

If $T$ is syntactically honest within data structures $d_1$ and $d_2$, then $T$ is a SYNIMPL tree transducer of $d_1$ by $d_2$.

**Proof:** Let $R(n,m)$ be the following proposition:
In order to prove that $R(n,m)$ is true for every $n$ and $m$, we prove the following:

(*1) prove that for every $m$, $R(0,m)$ is true;

(*2) prove that for every $m$, $R(1,m)$ is true;

(*3) prove that for every $m$, $R(n,m)$ implies $R(n+1,m)$.

(*1) $R(0,m)$ is true for every $m$

In order to prove that $R(0,m)$ is true for every $m$, we have to prove

(*1.1) $R(0,1)$ and $R(0,2)$ are true; and

(*1.2) $R(0,m)$ implies $R(0,m+1)$

(*1.1) $R(0,1)$ and $R(0,2)$ are true

Let $s \lessdot t$, $h_1(w)=s$, $|w|\leq 2$, $s \leftrightarrow s'$, and $t \leftrightarrow t'$, then we must show that $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. If $s'=t'$, then we are done. Suppose $s' \neq t'$.

Now $s \lessdot t$ implies that $s=t$; and $|w|\leq 2$ implies that $s$, $t \in SB$ and $SB$ is some element of the least element of $L(SBASE)$ where $SBASE=\{h_1(w)/|w|\leq 2\}$. From condition (iv) of Definition 6.14, letting $s=b_1$, $t=b_2$; and $b_1=c_1$, $b_2=c_2$; and $s'=d_1$, $t'=d_2$. Thus we must have $d_1,d_2 \in TB$, an element of the least element of $L(TBASE)$, where $TBASE=\{h_2(w)/|w|\leq 2\}$. Hence, by Theorem 3.41, we conclude that $\llbracket d_1 \rrbracket_{E_2} = \llbracket d_2 \rrbracket_{E_2}$; and this implies that $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. 
inductive step: $R(0,m)$ implies $R(0,m+1)$:

Let $s \overset{0}{\rightarrow} t$ (i.e., $s=t$), $h_1(w)=s$, $|w|=m$. Since $R(0,m)$ is true then if $s \neq s'$, $t \neq t'$ then $[s']_{E_2} = [t']_{E_2}$. Now using condition (vi) of Definition 6.14 let $A=h_1(v)$ and $|v|=m+1$ and $A \Rightarrow A_1 \Rightarrow \cdots \Rightarrow A_n$.

Let $s \Rightarrow t$, $h_1(v)=s$, $|v|=m+1$ and $A \Rightarrow A_1 \Rightarrow \cdots \Rightarrow A_n$. Then we will show that $[s']_{E_2} = [t']_{E_2}$. Now consider Figure 6.3. $R(0,m)$ implies

$$[s']_{E_2} = [t']_{E_2} \quad \text{(+1)}$$

Since $T$ is locally semiconsistent from source to target, then we must have

$$[B_1]_{E_2} = [s']_{E_2} \quad \text{(+2)}$$

$$[B_2]_{E_2} = [t']_{E_2} \quad \text{(+3)}$$

Comparing (+1), (+2), and (+3), we conclude that $[B_1]_{E_2} = [B_2]_{E_2}$. 

(*2) $R(1,m)$ is true for every $m$

For proving that $R(1,m)$ is true for every $m$, we have to prove

(*2.1) $R(1,1)$ and $R(1,2)$ are true; and

(*2.2) $R(1,m)$ implies $R(1,m+1)$.

(*2.1) $R(1,1)$ and $R(1,2)$ are true

Let $h_1(w)=2$, $|w|<2$, $s \Rightarrow t$, $e \in E_1$, $s \Rightarrow s'$, $t \Rightarrow t'$. Then we must show that $[s']_{E_2} = [t']_{E_2}$. Since $T$ is mono, thus we conclude that if $h_1(v)=t$ then $|w| \geq |v|$. $s \Rightarrow t$ implies that $s$, $t \in SB$, SB is an element.
Figure 6.3 Pictorial representation of $R(0,m+1)$
of the least element of $L(S_{BASE})$ where $S_{BASE} = \{ h(w) / |w| \leq 2 \}$. Now using Definition 6.14, letting $b_1 = c_1 = s$ and $b_2 = c_2 = t$; and $d_1 = s'$, $d_2 = t'$, and $d_1, d_2 \in T_{BASE}$, where $T_{BASE} = \{ h_2(w) / |w| \leq 2 \}$. Hence, by Theorem 3.41, we conclude that $\llbracket d_1 \rrbracket_{E_2} = \llbracket d_2 \rrbracket_{E_2}$. And since $s' = d_1$ and $t' = d_2$, thus $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$.

(*2.2) $R(1, m)$ implies $R(1, m+1)$

Let $h_1(w) = s$, $|w| = m+1$, $s = t$, $e \in E_1$, $s = s'$, $t = t'$, and $R(1, m)$ is true, then we must show that $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. Using Definition 3.3 we consider three cases.

(*2.2) Case (i)

$s = L$, $t = R$, and $e = (L = R) \in E_1$. Then we will have $L \rightarrow L'$, $R \rightarrow R'$. Since $T$ is syntactically honest, then $T$ must be locally semiconsistent from source to target, and this implies that $\llbracket L' \rrbracket_{E_2} = \llbracket R' \rrbracket_{E_2}$. Letting $s' = L'$ and $t' = R'$, we conclude that $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$.

(*2.2) Case (ii)

$s = t$, $e = (L = R) \in E_1$ where $s = L = t_1, \ldots, t_k$, $t = R = t_1, \ldots, t_k$. Hence, we must have:

$s' = L'[t'_1, \ldots, t'_k]$, $t' = R'[t'_1, \ldots, t'_k]$

where $t_i \neq t'_i$ for $i = 1, \ldots, k$. $(L = R) \in E_1$ implies that $\llbracket L \rrbracket_{E_1} = \llbracket R \rrbracket_{E_1}$. Since $T$ is syntactically honest, hence by condition (i), (iv) of Definition 6.14, we conclude that $\llbracket L' \rrbracket_{E_2} = \llbracket R' \rrbracket_{E_2}$. Using Lemma 3.18, we must have $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. 


\[(**2.2)** \text{Case (iii)}
\[
s = t, \ e = (L=R) \in E_1, \text{ and } t_j = t'_j \text{ where}
\]
\[s = f[t_1, \ldots, t_j, \ldots, t_k]
\[t = f[t_1, \ldots, t_j', \ldots, t_k]
\]
where \(fe \in \Sigma_1(w,a)\) and \(|w| = k\).

Now let \(s = s'\), \(t = t'\), then we must show that \([s']_E = [t']_E\). Several cases are possible:

\[(**2.2)** \text{case (iii) case (a)}
\[
\text{Let } \[\pi_1(g_1)\]_E = \[\pi_1(g_2)\]_E \text{ and}
\[
s = \pi_1(g_1)(h_1(w_1), \ldots, h_1(w_i), \ldots, h_1(w_r)), r < k
\]
\[t = \pi_1(g_2)(h_1(w_1), \ldots, h_1(w_i'), \ldots, h_1(w_r)), r < k
\]
where \(t_j\) is a subtree of \(h_1(w_i)\) and \(t'_j\) is a subtree of \(h_1(w'_i)\). Thus \(h_1(w_i) e > h_1(w'_i)\) and \(i \leq j\). Now, let \(s = s'\) and \(t = t'\), then we must have
\[
s' = \pi_2(g_1)(h_2(w_1), \ldots, h_2(w_i'), \ldots, h_2(w_r)),
\]
\[t' = \pi_2(g_2)(h_2(w_1), \ldots, h_2(w_i'), \ldots, h_2(w_r))
\]
Since \(R(1, m)\) is true and \(|w_i|, |w'_i| \leq m\), then by induction hypothesis we conclude that \([h_2(w_i)]_E = [h_2(w'_i)]_E\); and this implies that \(h_2(w_i) \geq x\) and \(h_2(w'_i) \geq x\). Now, let \(h_2(w_j) = v_j\) for \(j = 1, \ldots, r\); and \(h_2(w_i) = v'_i\).

Then \([s']_E =
\[
= \[\pi_2(g_1)(v_1, \ldots, v_i, \ldots, v_r)\]_E \text{ since } s' = \pi_2(g_1)(v_1, \ldots, v_i, \ldots, v_r)
\]
\[
= \[\pi_2(g_1)(v_1, \ldots, v_i, \ldots, v_r)\]_E \text{ since } v_i \geq x \text{ by induction}
\]
= [[\pi_2(g_1)](v_1, \ldots, v_i, \ldots, v_r)]_{E_2} \quad \text{since } v_i' \not\succ_X \text{ by induction}

= [[\pi_2(g_2)](v_1, \ldots, v_i, \ldots, v_r)]_{E_2} \quad \text{since } [[\pi_2(g_1)]_{E_2} = [[\pi_2(g_2)]_{E_2}

by condition (iv) of Definition 6.15.

(*2.2) Case (iii) case (b)

Suppose \( t_j \) and \( t'_j \) are not subtrees of \( h_1(w_i) \) for any \( i \). Thus, we must have:

\[
\begin{align*}
s &= \pi_1(P_1)(h_1(w_1), \ldots, h_1(w_r)), \quad s \not\succ s', \quad \pi_1(P_1) = f(..., t_j, ...)
s' &= \pi_1(P_1)(h_1(w_1), \ldots, h_1(w_r)), \quad t \not\succ t', \quad \pi_1(P_1) = f(..., t'_j, ...)
\end{align*}
\]

Since \( [[s]]_{E_1} = [[t]]_{E_1} \), then using Lemma 3.18, we must have \( [[\pi_1(P_1)]_{E_1} \]

\( = [[\pi_1(P_2)]_{E_2} \); and by using condition (iv) of Definition 6.14, we must have \( [[\pi_2(P_1)]_{E_2} = [[\pi_2(P_2)]_{E_2} \), hence, using Lemma 3.18, we conclude that

\[
[[s']]_{E_2} = [[\pi_2(P_1)(h_2(w_1), \ldots, h_2(w_r))]_{E_2}
= [[\pi_2(P_2)(h_2(w_1), \ldots, h_2(w_r))]_{E_2}
= [[t']]_{E_2}.
\]

(*3) \( R(n,m) \) implies \( R(n+1,m) \) for every \( m \)

Let \( s \in E_1 \), \( t, [[s]]_{E_1} = [[t]]_{E_1}, s \not\succ s', t \not\succ t', h_1(w) = s, |w| < m \). Then we must show that \( [[s']]_{E_2} = [[t']]_{E_2} \).
Now let $A 
ot\supset A'$.

Obviously, $\llbracket s \rrbracket_{E_1} = \llbracket A \rrbracket_{E_1} = \llbracket t \rrbracket_{E_1}$.

Since $R(n,m)$ is true, then we conclude that $\llbracket A' \rrbracket_{E_2} = \llbracket s' \rrbracket_{E_2}$. Now if we prove that $\llbracket t' \rrbracket_{E_2} = \llbracket A' \rrbracket_{E_2}$, then we are done.

Suppose $h_1(v) = A$. Since $E_1$ and $E_2$ are non-expanding, then $\text{SIZE}(s) \geq \text{SIZE}(A)$. Then since $T$ is syntactically honest, thus $T$ is a mono tree transducer, and this implies that $\vert w \vert \geq \vert v \vert$. Hence, $\llbracket A \rrbracket_{E_1} = \llbracket t \rrbracket_{E_1}$ implies $\llbracket A' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. Hence we conclude that $\llbracket s' \rrbracket_{E_2} = \llbracket t' \rrbracket_{E_2}$. □

Thus, we have shown that the least element (Theorem 3.41) of a finite set of elements of a data structure is needed to prove that the implementation tree transducer is SYNIMPL. It can be verified that removing any condition from syntactical honesty definition will cause a problem in proving that $T$ is a SYNIMPL tree transducer.

6.6.3. Development of SEMIMPL tree transducer

The definition of an "algebraic tree transducer" just concerns the structure issue, that is, it converts source tree into target tree. Concerning the structure issue, if $\llbracket t_1 \rrbracket_{E_1} = \llbracket t_2 \rrbracket_{E_1}$, and $T$ is a SYNIMPL tree transducer then $\llbracket t_1' \rrbracket_{E_2} = \llbracket t_2' \rrbracket_{E_2}$ where $t_1 \supset t_1'$ and $t_2 \supset t_2'$. Thus, we just map equivalent trees in to equivalent trees.
without checking that the source tree has the same meaning of the target tree. Covering both the syntactic and semantic issues, if \( t_1 \sigma = t_2 \sigma \) and \( T \) is a SEMIMPL tree transducer, then
\[
[t_1]_{E_1} = [t_2]_{E_1}
\]
and \( t_1 \) and \( t_2 \) have the same meaning if \( t_1 T t_1' \) and \( t_2 T t_2' \).

Thus, if \( T \) is a SEMIMPL tree transducer, then \( T \) converts elements of the source data structure into elements of the target data structure without changing the meaning of the elements of the source data structure. That is, we need tree transducers that preserve the semantics of source trees after transformation.

Our development follows that of Krishnaswamy and Strawn [31], who investigated conditions to be imposed on tree transducers to guarantee that they induced semantic-preserving translations. That is, if \( T \) is a semantic-preserving tree transducer and \( \text{TRANS}(T,s) \) is a family of tree transductions then \((t_1,t_2) \in \text{TRANS}(T,s)\) must imply that \( t_1 \) and \( t_2 \) have the same meaning. The following definition covers both the syntactic and semantic issues.

**Definition 6.16** [Semantic-Preserving Tree Transducer]

Let \( D_i=(G_i,A_i) \) and \( \Sigma_i \) be the signature of \( G_i \) for \( i=1,2 \).

Let \( T=(\gamma_1,\gamma_2) \) be a tree transducer such that \( \gamma_i=(\Sigma_i,\delta_i,n_i,\pi_i) \) for \( i=1,2 \) and \( \text{TRANS}(T,s)=(h_1(t),h_2(t))/h_1: T_{\Sigma_i} \rightarrow T_{\Sigma_i} \), and \( h_1 \) is a unique
homomorphism from $T_{\Sigma_i}$ to $T_{\Sigma_i}$ for $i=1,2$. Let $\mu_i^1: T_{\Sigma_i} \to A_1^i$ be the $\Sigma'_i$-homomorphism determined by the semantic homomorphism of $D_i$. The $T$ is "semantic-preserving" if $\mu_1^1 \circ \sigma_1(T_{\Sigma_1}) = \mu_2^1 \circ \sigma_2(T_{\Sigma_1})$; and $P_{A_1^i}(x) = P_{A_2^i}(x)$ for all $P \in \Sigma'(a,n)$ and for all $x \in \mu_1^1 \circ \sigma_1(T_{\Sigma_1})$.

It has been shown in [31] that semantic-preserving tree transducers induce a tree translation. This result is applicable in our context of data structure implementations to prove correct an implementation of a data structure. We investigate this in the next section. We will call "semantic-preserving" tree transducers as "SEMIMPL" tree transducers.

6.6.4. Correctness of SEMIMPL tree transducer

Informally, if $T$ is a SEMIMPL tree transducer for implementing a source data structure $d_1$ by a target data structure $d_2$, then $T$ is correct if $\text{TRANS}(T,s)$ determines a tree translation. That is, when a meaningful element of $d_1(t)$ is in input, it outputs a target element of $d_2$ that shares a meaning with $t$. We determine sufficient conditions under which it may be proved that SEMIMPL determines a tree translation.

Krishnaswamy and Pyster [30] have examined the correctness of semantic-syntax-directed translators. Here, we follow the work of [30] but extend the notion of tree transduction. In our work, equations will be used to extend the notion of tree transduction, and hence
to provide an additional capability to the tree transducers developed prior to this work. Considering the SEMIMPL tree transducer and combining the "translation" and "data structure", correctness of data structure implementation might be reduced to the correctness of translation.

We next define an "extended tree transduction" using equations which will be used not only in data structure implementation but also will be used in correctness of implementations.

**Definition 6.17 [Extended Tree Transduction]**

Let \( T=(Y_1, Y_2) \) be a tree transducer such that \( \gamma_i=(\Sigma', \Sigma_i, \eta_i, \pi_i) \) for \( i=1,2 \). Let \( E_1(E_2) \) be a set of source (target) equations associated with \( \Sigma_1(\Sigma_2) \). Let \( \text{TRANS}(\tau, s)=((h_1(t), h_2(t))/h_i: T_{\Sigma_i} \rightarrow T_{\Sigma_i} \) where \( h_i \) is a unique homomorphism from \( T_{\Sigma_i} \) to \( T_{\Sigma_i} \) for \( i=1,2 \). Then \( T \) along with \( E_1 \) and \( E_2 \) induces an "extended tree transduction" \( \text{ETRANS}(\tau, s, E_1, E_2)=\{(t_1, t_2)/ w \in T_s \) such that \( t_i \in [h_i(w)]_{E_i} \) for \( i=1,2 \). □

**Theorem 6.18**

Let \( d_1=(\Sigma_1, A_1, \emptyset) \) and \( d_2=(\Sigma_2, A_2, \emptyset) \) be specifications of free data structures. Let the associated tree transducer \( \tau=(\gamma_1, \gamma_2) \) be semantic-preserving such that \( \gamma_i=(\Sigma', \Sigma_i, \eta_i, \pi_i) \) for \( i=1,2 \). Then tree transduction of \( \tau \) (i.e., \( \text{TRANS}(\tau, s) \)) is a tree translation and \( \tau \) is a SEMIMPL of \( d_1 \) by \( d_2 \).

**Proof:** Let \( \text{TRANS}(\tau, s)=((h_1(t), h_2(t))/h_i: T_{\Sigma_i} \rightarrow T_{\Sigma_i} \) and \( h_i \) is a unique homomorphism for \( i=1,2 \). Then we need to prove that \( \gamma_1(h_1(t))=\gamma_2(h_2(t)) \) for
\((t)\) for the unique homomorphisms \(\nu_i: \Sigma_i \rightarrow A_i\) and for all \(t \in \Sigma_i^*\).

The proof is by induction on \(|t|\) (length of \(t\)) and we need to show that 
\[\tilde{\nu}_1(h_1(t)) = \tilde{\nu}_2(h_2(t))\] for all \(t \in \Sigma_i^*\). The full proof has been given in our earlier work \([36]\) and the proof will not be repeated here. \(\square\)

**Theorem 6.19**

Let \(d_1 = (\Sigma_1, A_1, E_1)\) and \(d_2 = (\Sigma_2, A_2, E_2)\) be some data structures. Let \(\gamma = (\gamma_1, \gamma_2)\) be a semantic-preserving tree transducer such that 
\(\gamma_i = (\Sigma_i, E_i, n_i, \pi_i)\) for \(i = 1, 2\); and \(\text{TRANS}(\gamma, s) = \{(h_1(t), h_2(t))/h_i: T_{\Sigma_i} \rightarrow T_{\Sigma_i}\}\) is a unique homomorphism for \(i = 1, 2\). Then the extended tree transduction of \(\gamma\), \(\text{ETRANS}(\gamma, s, E_1, E_2)\) is a tree translation. \(\square\)

Figure 6.4 illustrates graphical representation of Theorem 6.19.

6.7. An Illustration of Correctness of Syntactical Implementation of Data Structures

In this section, examples will be illustrated to show the actual concept of implementation of data structures. For this purpose, we have identified three examples: the first example illustrates the source data structure (DEQUEUE); the second example will illustrate the target data structure (ARRAY); and finally the third example will implement the source data structure by the target data structure.
Figure 6.4 Pictorial representation of Theorem 6.19

\[ g_i: T_{\Sigma_1} \rightarrow \llbracket T_{\Sigma_1} \rrbracket_{E_i}, \quad i=1,2 \]
6.7.1. Example 1: the source data structure (DEQUEUE)

Let \( DEQUEUE = (\Sigma_1, A_1, E_1) \) be a dequeue (double ended queue) data structure. A dequeue is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends of a dequeue LEFT and RIGHT.

Let \( N_1 = \{ DEQUEUE, DATA, BOOLEAN \} \), abbreviating \( DEQUEUE, DATA \) and \( BOOLEAN \) as \( Q, D, \) and \( B \). The major operations \( (\Sigma_1) \) on dequeue are defined in Figure 6.5.

We abbreviate CLEAR, INSERTLEFT, REMOVELEFT, INSERTRIGHT, REMOVERIGHT, RIGHTELEMENT, and EMPTY as CLR, INL, REL, INR, RER, RIG and EMP. Let \( A_1 = \langle A_1, F_1 \rangle \) where (intentionally we have dropped the subscripts) \( F_1 = \{ CLR, INL, REL, INR, RER, RIG, EMP \} \). In order to deal precisely with the underlying elements of \( A_1 \), consider the modeling domain of dequeues of length \( n \) (\( n > 0 \)) to be finite sequences denoted \( <d_1, \ldots, d_n> \) where \( d_1 \) is the left (rear) element and \( d_n \) is the right (front) element. Empty dequeue will be denoted \(<>\). Using this concept we can define the operations on the data structure DEQUEUE as in Figure 6.5. For clarity we have dropped all of the subscripts \( (A_1) \) from the function descriptions.

To complete definition of DEQUEUE, Figure 6.6 gives a sufficiently complete semantic definition of data structure DEQUEUE.
signature Σ₁

Σ₁(λ, Q) = {CLEAR, ERROR}
Σ₁(QD, Q) = {INSERTRIGHT, INSERTLEFT}
Σ₁(Q, Q) = {REMOVERIGHT, REMOVELEFT}
Σ₁(Q, D) = {RIGHTELEMENT}
Σ₁(Q, B) = {EMPTY}
Σ₁(λ, D) = {d}

definition of functions in algebra A₁

CLEAR( ) = <>
INSERTRIGHT(<d₁,...,dₙ>, d) = <d₁,...,dₙ,d>
INSERTLEFT(<d₁,...,dₙ>, d) = <d,d₁,...,dₙ>
REMOVERIGHT(<d₁,...,dₙ₋₁,dₙ>) =
  {<d₁,...,dₙ₋₁> if n>0
   ERROR otherwise

REMOVELEFT(<d₁,d₂,...,dₙ>) =
  {<d₂,...,dₙ> if n>0
   ERROR otherwise

RIGHTELEMENT(<d₁,...,dₙ>) =
  {dₙ if n>0
   UNDEFINED otherwise

EMPTY(<d₁,...,dₙ>) =
  {TRUE if n=0
   FALSE otherwise

Figure 6.5 Specification of DEQUEUE data structure
(1): \( \text{RER}[\text{CLR}] = \text{ERROR} \)
(2): \( \text{REL}[\text{CLR}] = \text{ERROR} \)
(3): \((X_1 = \text{CLR}) \Rightarrow \text{RER}[\text{INL}[X_1X_2]] = \text{CLR} \)
(4): \( \text{not} (X_1 = \text{CLR}) \Rightarrow \text{RER}[\text{INL}[X_1X_2]] = \text{INL}[\text{RER}[X_1X_2]] \)
(5): \((X_1 = \text{CLR}) \Rightarrow \text{REL}[\text{INR}[X_1X_2]] = \text{CLR} \)
(6): \( \text{not} (X_1 = \text{CLR}) \Rightarrow \text{REL}[\text{INR}[X_1X_2]] = \text{INR}[\text{REL}[X_1X_2]] \)
(7): \( \text{RER}[\text{INR}[X_1X_2]] = X_1 \)
(8): \( \text{REL}[\text{INL}[X_1X_2]] = X_1 \)
(9): \( \text{EMP}[\text{CLR}] = \text{TRUE} \)
(10): \( \text{EMP}[\text{INR}[X_1X_2]] = \text{FALSE} \)
(11): \( \text{EMP}[\text{INL}[X_1X_2]] = \text{FALSE} \)
(12): \( \text{RIG}[\text{CLR}] = \text{UNDEFINED} \)
(13): \((X_1 = \text{CLR}) \Rightarrow \text{RIG}[\text{INR}[X_1X_2]] = X_2 \)
(14): \( \text{not} (X_1 = \text{CLR}) \Rightarrow \text{RIG}[\text{INL}[X_1X_2]] = \text{RIG}[X_1] \)
(15): \( \text{RIG}[\text{INR}[X_1X_2]] = X_2 \)
(16): \( \text{INR}[\text{INL}[X_1X_2]X_3] = \text{INL}[\text{INR}[X_1X_3]X_2] \)
(17): \( \text{INL}[\text{INR}[X_1X_2]X_3] = \text{INR}[\text{INL}[X_1X_3]X_2] \)
(18): \( \text{REL}[\text{INR}[\text{INL}[X_1X_2]X_3]] = \text{INR}[X_1X_3] \)
(19): \( \text{RER}[\text{INR}[\text{INL}[X_1X_2]X_3]] = \text{INL}[X_1X_3] \)
(20): \( \text{INR}[\text{CLR} X_1] = \text{INL}[\text{CLR} X_1] \)

Figure 6.6 Equational specification of DEQUEUE data structure
6.7.2. Example 2: the target data structure (ARRAY)

Let $\text{ARRAY}=(\Sigma_2,A_2,E_2)$ be an array data structure. For arrays we are concerned with only two operations which retrieve and store values.

Let $A_2=<A_2,F_2>$ where (we intentionally have dropped the subscripts $(A_2)$).

$F_2=$\{$\text{UNDEFINED, NEWARRAY, UNDERFLOW, SUCC, PRED, ASSIGN, ACCESS, LEFT, RIGHT}$.\}

Let $N_2=$\{$\text{ARRAY, INTEGER, RANGE, BOOLEAN}=$\{$A,I,R,B$.\}. Then the operations $(\Sigma_2)$ on array are defined in Figure 6.7. The semantics of an array will be a set $AR\subseteq\text{INT} \times \text{INT}$ where $\text{INT}$ denotes the set of integer numbers. $AR$ has a property that $(x,y)\in AR$ and $(x,y')\in AR$ implies $y=y'$. Thus if array $AR$ was $\{(x_1,y_1),\ldots,(x_n,y_n)\}$ then it means that $AR(x_i)=y_i$ for $i=1,\ldots,n$; and $AR$ is undefined elsewhere, where $x_i,y_i\in\text{INT}$. The cardinality of $AR$ is unbounded. That is, we are using arbitrary large size arrays. Following this concept, we can define the operations on the data structure $\text{ARRAY}$ as in Figure 6.7. Note that, in description of the functions in $F_2$ we have left out the subscripts $A_2$.

We abbreviate NEWARRAY, ASSIGN, ACCESS, LEFT, RIGHT, and ZERO as NEW, ASN, ACC, LFT, RIT, and ZER. Equational specification of an ARRAY data structure is given in Figure 6.8

6.7.3. Example 3: implementation of DEQUEUE by ARRAY

A DEQUEUE data structure can be implemented in terms of an ARRAY data structure. Each "dequeue" value is a structure with an "array", 
signature $\Sigma_2$

$\Sigma_2(\lambda, R) = \{\text{UNDEFINED}, d\}$

$\Sigma_2(\lambda, A) = \{\text{NEWARRAY}, \text{UNDERFLOW}\}$

$\Sigma_2(\lambda, R) = \{\text{ASSIGN}\}$

$\Sigma_2(\lambda, R) = \{\text{ACCESS}\}$

$\Sigma_2(\lambda, I) = \{\text{LEFT}, \text{RIGHT}, \text{ZERO}, \text{ONE}\}$

$\Sigma_2(\lambda, I) = \{\text{SUCC}, \text{PRED}\}$

$\Sigma_2(\lambda, I) = \{\text{EQ}\}$

definitions of functions in algebra $A_2$

$\text{ASSIGN}(((x_1, y_1), \ldots, (x_n, y_n)), \text{index}, \text{value}) =$

$\left\{ \begin{array}{ll}
((x_1, y_1), \ldots, (x_n, y_n)) \cup \{(\text{index}, \text{value})\} & \text{if there exists a unique } x_i \text{ such that } \text{index} = x_i; \text{ or}

((x_1, y_1), \ldots, (x_n, y_n)) \cup \{(\text{index}, \text{value})\} & \text{if } x_i \text{'s are distinct and } x_i \neq \text{index} \text{ for } i=1, \ldots, n; \text{ or}

\text{error} & \text{if there exists } x_i = x_j = \text{index for } i \neq j
\end{array} \right.$

$\text{ACCESS}(((x_1, y_1), \ldots, (x_n, y_n)), \text{index}) =$

$\left\{ \begin{array}{ll}
y_i & \text{if there exists a unique } x_i = \text{index}

\text{UNDEFINED} & \text{otherwise}
\end{array} \right.$

$\text{NEWARRAY}() = \{\}$

$\text{PRED}(x) = x - 1$

$\text{SUCC}(x) = x + 1$

$\text{EQ}(x_1, x_2) =$

$\left\{ \begin{array}{ll}
\text{TRUE} & \text{if } x_1 = x_2

\text{FALSE} & \text{otherwise}
\end{array} \right.$

Figure 6.7 Specification of ARRAY data structure
(1): ACC[NEW X₁] = UNDEFINED
(2): (X₂ = X₄)→ACC[ASN[X₁X₂X₃]X₄] = X₃
(3): not (X₂ = X₄)→ACC[ASN[X₁X₂X₃] X₄] = ACC[X₁X₄]
(4): EQ[ZERO ZER] = TRUE
(5): EQ[SUCC[X₁]SUCC[X₂]] = EQ[X₁X₂]
(6): SUCC[PRED[X₁]] = X₁
(7): PRED[SUCC[X₁]] = X₁
(8): SUCC[ZERO] = ONE

Figure 6.8 Equational specification of ARRAY data structure
whose two indexes of array have especial meaning. The array element with index of LEFT and RIGHT will point to the rear element and the front element of dequeue respectively. For instance, if <d_3d_2d_1> is a value for a dequeue then the LEFT is pointing to d_3 (rear) and RIGHT is pointing to d_1 (front).

The implementation of the DEQUEUE data structure with the ARRAY data structure is given in Figure 6.9. Also, in [46], using Pascal programming language, they have given implementation of deques using arrays. In Figure 6.9 we have left out the domains of source and target trees.

We now illustrate a representation of deques using arrays. For instance, the empty dequeue is shown in Figure 6.10(a), the dequeue with just one element (<d>) is represented as shown in Figure 6.10(b), and the dequeue <d_1d_2d_3> is represented as shown in Figure 6.10(c). Thus, the dequeue is empty if and only if (LEFT+1=RIGHT).

The implementation tree transducer is syntactically honest. To prove that this tree transducer is syntactically honest, we need to prove conditions (i), (ii), (iii), (iv), (v), and (vi) of Definition 6.14.

Condition (i) of Definition 6.14

Here, we need to show that if (L=R) ∈ E_1, L_1R_1', R_2R_2', then \([L']_E_2 = [R']_E_2\). For illustration, we will just show that equation
\[ d \leftarrow 0: (\lambda, P) \rightarrow d \]

\[
\begin{align*}
\text{REL}[INL[CLR X_1]] & \rightarrow 1: (P, q) \\
\text{RER}[INR[CLR X_1]] & \rightarrow 2: (P, q) \\
\text{REL}[INR[CLR X_1]] & \rightarrow 3: (P, q) \rightarrow \text{ASN[ASN[NEW LFT ZERO]RIT ONE]} \\
\text{RER}[INL[CLR X_1]] & \rightarrow 4: (P, q) \\
\text{CLR} & \rightarrow 5: (\lambda, q) \\
\text{RER[CLR]} & \rightarrow 6: (\lambda, q) \rightarrow \text{UNDERFLOW} \\
\text{REL[CLR]} & \rightarrow 7: (\lambda, q) \\
\text{REL[X_1]} \rightarrow 8: (q, q) \rightarrow \text{ASN[X_1 ACC[X_1 LFT]SUCC[ACC[X_1 LFT]]]} \\
\text{RER[X_1]} \rightarrow 9: (q, q) \rightarrow \text{ASN[X_1 ACC[X_1 RIT]PRED[ACC[X_1 RIT]]]} \\
\text{EQ[X_1]} \rightarrow 10: (q, b) \rightarrow \text{EQ[ACC[X_1 RIT]SUCC[ACC[X_1 LFT]]]} \\
\text{INL[X_1 X_2]} \rightarrow 11: (qP, q) \rightarrow \text{ASN[t_1 t_2 t_3]} \quad \text{where} \\
& \quad t_1 = \text{ASN[X_1 ACC[X_1 LFT]X_2]} \\
& \quad t_2 = \text{ACC[X_1 LFT]} \\
& \quad t_3 = \text{PRED[ACC[X_1 LFT]]} \\
\text{INR[X_1 X_2]} \rightarrow 12: (qP, P) \rightarrow \text{ASN[w_1 w_2 w_3]} \quad \text{where} \\
& \quad w_1 = \text{ASN[X_1 ACC[X_1 RIT]X_2]} \\
& \quad w_2 = \text{ACC[X_1 RIT]} \\
& \quad w_3 = \text{SUCC[ACC[X_1 RIT]]} \\
\eta_1(q) = Q & \quad \eta_2(q) = A \\
\eta_1(P) = D & \quad \eta_2(P) = R \\
\eta_1(b) = B & \quad \eta_2(b) = B
\end{align*}
\]

Figure 6.9 Implementation tree transducer T
(a) an empty dequeue (<>):

![Diagram of an empty dequeue](image1)

(b) a dequeue with one element (<d>):

![Diagram of a dequeue with one element](image2)

(c) a dequeue with three elements (<d_1,d_2,d_3>):

![Diagram of a dequeue with three elements](image3)

Figure 6.10. Dequeue representation
(3) of $E_1$ is satisfying the local semiconsistency condition. Thus, we have to show that

$$
RER_{[\text{LIN}[\text{CLR} \times_1]]} \Rightarrow \text{L'}; \text{ and } \text{CLR} \Rightarrow R'
$$

implies $[[\text{L'}]]_{E_2} = [[\text{R'}]]_{E_2}$. By applying $T$, we will have:

- $L' = \text{ASN}[\text{NEW LFT ZER} \text{RIT ONE}]$;
- $R' = \text{ASN}[\text{NEW LFT ZER} \text{RIT ONE}]$.

Thus, $[[L']]_{E_2} = [[R']]_{E_2}$. In like manner, it can be proved that the other equations of $E_1$ satisfy the conditions of local semiconsistency.

**Condition (ii) of Definition 6.14**

Figure 6.4 and Figure 6.6 are proofs of this condition.

**Condition (iii) of Definition 6.14**

$$
\xi_1 = h_1(w_1), \quad \xi_2 = h_1(w_2), \quad \text{SIZE}(\xi_1) \geq \text{SIZE}(\xi_2) \text{ implies } |w_2| \geq |w_1|.
$$

The proof of this condition emerges from $E_1, E_2,$ and $T$.

**Condition (iv) of Definition 6.14**

$$
\text{SBASE} = \{\text{INR[CLR d], INL[CLR d], ...} \}
$$

$$
\text{TBASE} = \{\text{ASN}[w_1, w_2, w_3], \text{ASN}[t_1, t_2, t_3], ... \}
$$

where

- $w_1 = \text{ASN}[X \text{ ACC}[X \text{ RIT}]d]$,
- $w_2 = \text{ACC}[X \text{ RIT}]$,
- $w_3 = \text{SUCC}[\text{ACC}[X \text{ RIT}]]$,
- $X = \text{ASN}[\text{NEW LFT ZER} \text{RIT ONE}]$,
- $t_1 = \text{ASN}[X \text{ ACC}[X \text{ LFT}]d]$,
- $t_2 = \text{ACC}[X \text{ LFT}]$,
- $t_3 = \text{PRED}[\text{ACC}[X \text{ LFT}]]$.
Let \( b_1^* = \text{INR} \{ d \} \), \( b_2^* = \text{INL} \{ d \} \), \( c_1^* = \{ d \} \), \( c_2^* = \text{ASN} \{ t_1, t_2, t_3 \} \). Obviously, using equation (20) of Figure 6.4, 
\[ \left[ b_1^* \right]_{E_1} = \left[ b_2^* \right]_{E_1} \]. We have to show that \( \left[ d_1^* \right]_{E_2} = \left[ d_2^* \right]_{E_2} \). It can be proved that \( d_1^* \xrightarrow{\ast} m \), \( d_2^* \xrightarrow{\ast} m \) where \( m \in T \). Similarly, this condition can be proved for other elements in SBASE and TBASE.

**Condition (v) of Definition 6.14**

From Figure 6.7 construct
\[
\text{SOURCE} = \{ \pi_1(P) \mid P \in \Sigma' \} \quad \text{and} \quad \text{TARGET} = \{ \pi_2(P) \mid P \in \Sigma' \}.
\]

For simplicity, let us use \( P' \) for \( \pi_1(P) \) and \( P'' \) for \( \pi_2(P) \). Then the least elements of \( L(\text{SOURCE}) \) and \( L(\text{TARGET}) \) will be \( L' \) and \( L'' \) respectively.

\[
L' = \{ \{1', 2', 3', 4', 5'\}, \{6', 7'\}, \{8'\}, \{9'\}, \{10'\}, \{11'\}, \{12'\} \}
\]

\[
L'' = \{ \{1''\}, \{6''\}, \{8''\}, \{9''\}, \{10''\}, \{11''\}, \{12''\} \}
\]

Now, having \( L' \) and \( L'' \), it can easily be verified that \( L' \) and \( L'' \) are satisfying the condition (iv) of Definition 6.15.

Now, let \( S = \{1', 2', 3', 4', 5'\} \), \( a = 1' \), \( c = 2' \). \( (a, 1'') \in T \), let \( T = \{1''\} \).

Now \( (2', 1'') \in T \). Now since \( 1'' \in T \), thus, we are done.

**Condition (vi) of Definition 6.14**

This condition can be proved by using the concept of \( E_1, E_2 \), and \( T \).

Thus, implementation tree transducer \((T)\) is syntactically honest.

Hence, by Theorem 6.14, \( T \) is a SYNIMPL tree transducer.
6.8. Summary, Conclusions

We have presented an algebraic framework for data structures, and studied four important aspects of data structures, namely specification, equivalence, implementation, and implementation correctness. Within this framework, we made a clear distinction between a data structure specification and its implementation. Also, we used the concept of equivalency to prove that conditional equations are "more powerful" than simple equations.

A general technique for the implementation of data structures developed. That is, given data structures $d_1$ and $d_2$, an implementation of $d_1$ by $d_2$ is defined separately on the syntactical and semantical levels of data structure elements.

One of the key results of this chapter is the development of syntactically honest tree transducers. This development is based on a lattice developed in Chapter 3.
7. CONCLUSIONS AND FUTURE WORK

This thesis primarily dealt with developing a framework within which we could discuss the implementation of abstractly specified data structures. An emphasis of this thesis was to develop criteria for correctness of implementation. To this end, it was necessary for us to develop a theory within which data structures may be specified and implemented. As the specification of data structures is intended to be implementation independent, we develop an algebraic theory for our framework. It turns out that this theory is adequate not only to specify a data structure abstractly, but also to determine the implementation of one data structure by another. We give several examples of algebraic specification and implementation of data structures.

The algebraic framework provided a base from which many important results and questions arose about data structures. Some of these questions have been addressed in this thesis, while others suggest directions for further research. Below, we first summarize the conclusions and results of our work, and then indicate how this work may be further extended.

7.1. Results and Summary

We have investigated an algebraic model to specify and implement data structures, and developed a framework to analyze the correctness of implementations. In our approach, the specification of data struc-
tures is by initial algebra semantics and algebraic equations. We use tree transducers to implement data structures, reducing the problem of implementation to a problem of translation. This work makes a sharp distinction between the specification of a data structure and its implementation.

This thesis reduces the problem of implementation of data structures to a problem of translation of languages by the following approach. Let D be a data structure to be implemented in a language L. Let L' be a new language comprising all of L and primitives for the data structure D. Then a translation from L' to L determines an implementation of D in L. To this end we give a formal definition of data structures on an abstract level. The definition of data structures enables us to define their implementation. We then address the issue of correctness of the implementation on two levels: syntactical and semantical. Syntactically correct implementations deal with the algebraic equations that specify a data structure, while the semantically correct implementations define correctness on the basis of the semantic algebra in the data structure specifications. All the implementations, however, are specified by tree transducers. The issue of the tree transducers is addressed on the syntactic and semantic levels as described informally above.

Two key syntactical properties of tree transducers have been investigated in this thesis, the "consistency" and "semiconsistency"
of tree transducers with respect to the algebraic equations defining the source and target data structures. In our work, we need these properties for the syntactical correctness proof of the implementation of data structures.

We provide several illustrations of our techniques for data structure specification and implementation. The examples that are examined in this thesis include set, binary tree, queue, dequeue, and an array. In particular, an example of an implementation of a dequeue data structure by an array data structure has been provided in full. The criteria developed for syntactically correct implementations of data structures are used to prove this implementation correct.

One of the key steps required to develop the criteria for correct implementations of data structures was to determine the effect of equations used in their specification. It was necessary to apply lattice theoretic techniques to determine when instances of a data structure formed by apparently different sequences of operations were in fact equivalent. It was shown that a set of operations and composite operations on a data structure could be partitioned by lattice theoretic techniques into equivalence classes of operations and composite operations which were "equivalent". It is this partition that is crucial in developing the criteria for syntactically correct implementations of data structures.
7.2. Directions for Future Research

This thesis raises several open questions and points to research directions in the area of provably correct implementations of data structures. One of the key results of this work is the development of criteria to determine whether an implementation is correct with respect to specification of source and target data structures. A set of sufficient conditions is provided to determine the correctness. It would be useful to develop an algorithm for the construction of a tree transducer which is provably correct implementation of the source data structure by the target data structure.

We used "applicative" languages for the specification and implementation of data structures. This is because algebraic methods are more amenable to applicative languages. This work does not examine the implementation of data structures in "nonapplicative" languages. The work presented here can be complemented and extended by analyzing data structures in nonapplicative and hybrid (applicative and nonapplicative) languages.
8. BIBLIOGRAPHY


9. ACKNOWLEDGEMENTS

The author is grateful to the members of his committee for their patient guidance and assistance with his research. He is particularly grateful to his thesis adviser, Dr. Ramachandran Krishnaswamy (now with Ford Aerospace and Communications, Palo Alto, CA), who has spent many hours sharing his understanding of data structures, algebraic equations, and tree transducers, analyzing his ideas, and making valuable suggestions which have improved the presentation of his results. Dr. Krishnaswamy's guidance and example as a scientist have directed my development in the field of computer science, and the patience and confidence in me that he has shown have made this development possible.

The author is also grateful to his advisor, Dr. George Strawn, for his valuable guidance and suggestions and also for his kind consent to serve on the author's committee. He is also indebted to Mrs. Lucille Kilmer for typing an exceptionally difficult manuscript quickly and accurately.

The understanding and support of Dr. Robert Stewart, and the financial support provided by the Department of Computer Science at Iowa State University have gone a long way in helping the author complete his research.

Finally, the author would like to thank his parents, Bagher Parsian and Monireh Parsian, and his brother and sister, Ahmad Parsian and Naier A. Parsian, who made it all possible.