Simulation and Transformation Techniques for Rewrite Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op dinsdag 19 december 1995 te 10:30 uur

door

Jan Cornelis VERHEUL,

informatica ingenieur,

geboren te Papendrecht.
Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. J. van Katwijk,

Toegevoegd promotors:
Dr. P.G. Kluit.

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter,
Prof. dr. ir. J. van Katwijk, (Technische Universiteit Delft) promotor,
Dr. P.G. Kluit, (Technische Universiteit Delft) toegevoegd promotors,
Prof. dr. P. Klint (Universiteit van Amsterdam),
Dr. E.M. Paalvast (Bakkenist Management Consultants B.V.),
Prof. dr. ir. H.J. Sips, (Technische Universiteit Delft, Universiteit van Amsterdam),
Prof. dr. S.C. van Westrhenen, (Technische Universiteit Delft),
Prof. dr. Winterstein, (FHT Mannheim).

This research has been sponsored by TNO Delft via the ParTool project.
Contents

1 Introduction ................................................. 1

2 Basic Notions ............................................... 9
  2.1 Introduction ........................................... 9
  2.2 The Term Concept ..................................... 9
  2.3 Term Analysis ........................................ 13
  2.4 Semantics of Terms and Signatures ................ 17

3 The Simulation Concept ................................ 21
  3.1 Introduction ........................................... 21
  3.2 Simple Step Simulation ............................... 22
  3.3 Class Step Simulation ................................ 23
  3.4 Composed Step Simulation ............................ 25
  3.5 Simulations of Term Rewriting Systems .......... 27
  3.6 Composition of Different Simulation Techniques .. 29

4 Resolution of Overloadedness ......................... 31
  4.1 Introduction ........................................... 31
  4.2 Resolution by Class Step Simulation ............... 31
  4.3 Resolution by Composed Step Simulation .......... 35
  4.4 A Large Example ..................................... 38

5 Retrenched Vocabularies ................................. 43
  5.1 Introduction ........................................... 43
  5.2 The Formal Definition ................................ 44
  5.3 Term Analysis for Vocabularies .................... 48
  5.4 Relations between Vocabularies and Signatures ... 56
    5.4.1 Introduction .................................... 56
    5.4.2 The Derivation of Vocabularies from Signatures 56
    5.4.3 The Derivation of Signatures from Vocabularies 57
    5.4.4 Properties of Vocabularies Derived from Signatures 59
  5.5 Relations between Vocabularies and Grammars ... 60
    5.5.1 Introduction .................................... 60
    5.5.2 The Conversion of Grammars to Vocabularies ... 61
5.5.3 The Conversion of Vocabularies to Grammars

6 Elementary Transformations

6.1 Introduction

6.2 Informal Description of Transformations

6.2.1 Introduction

6.2.2 Identity Transformations

6.2.3 Structure Transformations

6.3 Formal Descriptions of Transformations

6.3.1 Introduction

6.3.2 Assumptions and Pre-conditions

6.3.3 Derivation of New Vocabulary

6.3.4 Definition of Domain Mapping

6.3.5 Derivation of Rewrite System

6.4 Leaf Side Distinguishing

6.5 Root Side Distinguishing

6.6 Root Position Distinguishing

6.7 Leaf Side Identification

6.8 Root Side Identification

6.9 Root Position Identification

6.10 Contraction

6.11 Expansion

6.12 Correctness

7 Semantic Aspects of Transformations

7.1 Introduction

7.2 Fundamental Abstraction Mechanisms

7.3 Consequences for Operand and Result Types

7.3.1 Introduction

7.3.2 Leaf Side Distinguishing

7.3.3 Root Side Distinguishing

7.3.4 Root Position Distinguishing

7.3.5 Leaf Side Identification

7.3.6 Root Side Identification

7.3.7 Root Position Identification

7.3.8 Contraction

7.3.9 Expansion

7.4 Simplifying Type Systems

7.5 Reinterpreting Operations

7.5.1 Introduction

7.5.2 Leaf Side Distinguishing

7.5.3 Root Side Distinguishing

7.5.4 Root Position Distinguishing

7.5.5 Leaf Side Identification
CONTENTS

7.5.6 Root Side Identification ................................................. 132
7.5.7 Root Position Identification ............................................. 133
7.5.8 Contraction .............................................................. 134
7.5.9 Expansion ............................................................... 134
7.5.10 Simplifying Type Systems ............................................. 135

8 The Transformation Space .................................................. 137
8.1 Introduction ............................................................... 137
8.2 Basic Definitions ........................................................ 138
8.3 Restructuring Transformations ........................................... 145
  8.3.1 Introduction .......................................................... 145
  8.3.2 The Reversal of Contractions ....................................... 145
  8.3.3 Formal Definition of Restructuring Transformations .......... 148
  8.3.4 Justification for Design Decisions .................................. 151
8.4 Generalized Identity Transformations ................................ 152

9 Transformation Strategies ............................................... 155
9.1 Introduction ............................................................. 155
9.2 Identical Domain Transformations ..................................... 157
  9.2.1 Introduction ......................................................... 157
  9.2.2 Leaf Side Generalization .......................................... 157
  9.2.3 Root Side Generalization .......................................... 159
  9.2.4 Leaf Side Specialization .......................................... 161
  9.2.5 Root Side Specialization .......................................... 163
9.3 Rule Reductions .......................................................... 164
  9.3.1 Introduction ......................................................... 164
  9.3.2 Enabling Generalizations .......................................... 166
  9.3.3 Enabling Identifications .......................................... 170
  9.3.4 Enabling Distinguishing ........................................... 175
  9.3.5 Reducing Rule Increases .......................................... 176
  9.3.6 A Simple Strategy .................................................. 178
  9.3.7 Practical Benefits of Rule Reductions ........................... 180
9.4 Size Reductions .......................................................... 182
  9.4.1 Introduction ......................................................... 182
  9.4.2 Size Reductions Versus Vocabulary Reductions ............... 182
  9.4.3 Size Reductions Versus Rule Reductions .......................... 183
  9.4.4 Practical Benefits of Size Reductions ........................... 184
9.5 Vocabulary Reductions ................................................... 184
  9.5.1 Introduction ......................................................... 184
  9.5.2 A Simple Strategy .................................................. 184
  9.5.3 Practical Benefits of Vocabulary Reductions .................. 186
9.6 Acquiring Special Properties ........................................... 186
  9.6.1 Introduction ......................................................... 186
  9.6.2 Acquiring Integratedness ......................................... 186
CONTENTS

9.6.3 Acquiring Satisfiability ........................................ 187
9.6.4 Acquiring the Trivial Combination Free Property ........ 187
9.6.5 Normalization .................................................. 188
9.6.6 Acquiring the Uniquely Starting Property ................. 188
9.6.7 Acquiring Modular Decomposability ....................... 189
9.6.8 Acquiring Properties with Regard to Arity ............... 191

10 Case ........................................................................ 193
10.1 Introduction ....................................................... 193
10.2 Static Semantics of Pascal ....................................... 194
10.3 The Specification Language ..................................... 196
10.4 The Optimizations ................................................ 199
  10.4.1 Introduction ................................................... 199
  10.4.2 Rule Reductions ............................................... 200
  10.4.3 Size Reductions ............................................... 205
  10.4.4 Vocabulary Reductions ...................................... 210
  10.4.5 Acquiring Special Properties .............................. 210
10.5 Conclusion .......................................................... 213

11 Concluding Remarks ................................................ 215
11.1 Introduction ........................................................ 215
11.2 Generalizing Ground Rewriting ............................... 215
11.3 Adding Conditions to Rules .................................... 218
11.4 More Sophisticated Optimization Strategies ............... 220
11.5 The Necessity of the Retrenched Vocabulary Formalism .. 220
11.6 Implementing the Theory ....................................... 221
11.7 The Research Question ......................................... 222
Acknowledgements

The research for this thesis was performed at the Delft University of Technology and was funded by TNO Delft via the ParTool project. I wish to thank a number of people who had a pronounced influence on the course of the project. First and foremost this was Peter Kluit, who gave me critical and inspiring feedback on each of my new ideas. I am much indebted to Jan van Katwijk, my “promotor”, and to Edwin Paalvast, who gave me useful comments at regular times. I want also to mention René Dekker, my colleague and room mate, with whom I had many discussions on the research subject, but also on interesting other subjects. Thanks to so many people whom I met only once or twice on conferences and other events and who provided me useful suggestions for improvements and extensions. Finally I wish to thank leading researchers in the field who taught me the basics about term rewriting, termination, confluence, algebras, typing, etc. in their interesting articles and books. This research project has been successful thanks to amongst others my tenor saxophone. This faithful piece of brass enabled me to release frustrations at critical moments.
Chapter 1

Introduction

In recent years subtree replacement techniques have become increasingly popular for the transformation of tree structured data, witness the development of various systems (tools) that implement this subtree replacement technique in one way or another. To mention just a few examples: TXL [13], ASF+SDF [28], Trafola [27], Rule Language [9]. This subtree replacement technique, also known under the name “rewriting technique” or “template replacement technique”, allows one to specify transformation processes on tree structured data in a declarative way [26]. Concrete instances of tree structured data are: intermediate representations of computer programs, representations of data in databases, mathematical formulae, etc. It is a known fact that virtually all forms of data can be represented efficiently as trees [60, 52].

A formalization of transformation processes based on subtree replacement techniques is desirable because of the ability to make provable assertions about termination of transformation processes, length of transformation processes in case of termination, well-definedness of transformation results: determinism of the entire process, which may consist of non-deterministic transformation steps, and many other properties of rule based transformation systems. The formalism of term rewriting systems is suitable for the formal foundation of subtree replacement techniques because it is in fact the subtree replacement technique in its most pure and simple form. Unfortunately a formalization based on term rewriting is hampered or prevented altogether in practice because there is a fairly large gap between the theory that is available in the field of term rewriting systems and what is demanded in practical transformation cases.

Research in the field of term rewriting systems has, amongst others, the objective to bridge this gap between formal possibilities and practical demands. Since the publication of the well-known paper of Knuth and Bendix in 1970 [43], which formed the basis of a great many later papers on the term rewriting subject, three main directions can be distinguished to make the basic term rewriting principle better suited for practical problems:

- Extension of the basic rewriting principle, i.e. enriching the basic rewriting principle with additional primitives. Examples of this direction are:
– conditional term rewriting [37, 7],
– typed term rewriting [54, 54],
– priority rewriting [5, 48],
– rewriting modulo a set of equations [36, 4],
– higher order rewriting [49].

• Restriction of the rewrite principle in one way or another, i.e. restricting the
term domain and/or the rule domain to subsets of the original domain in order
to attain certain desired properties. Examples of this direction are:

– orthogonal (= left linear, non-overlapping) rewrite systems [40, 42],
– rewrite systems that follow the constructor discipline [16, 47],
– combinator reduction systems [49],
– string rewriting systems [8, 34].

• Development of theory to support the rewriting principle, i.e. theory to make
formal assertions about rewrite processes or to ease making certain formal
assertions about rewrite processes. Examples of this direction are:

– theory to establish confluence of rewrite systems [51, 39],
– theory to establish termination (also called strong normalization) of re-
write systems [15, 14, 44],
– theory about turning a set of equations into a confluent and terminating
set of rewrite rules [43, 3],
– theory about modular properties of rewrite systems [58, 53],
– theory about (normalizing) reduction strategies [59, 56].

Neither of the lists is meant to be complete. Almost each combination of exten-
sions and/or restrictions of the first and/or the second list, and theoretical aspects
of the third list raises interesting research questions which have been answered more
or less satisfactory in the past, or for which active research is being done in the
present. In fact an extension or a restriction of the rewrite principle is only inter-
esting if it is possible to develop useful theory for it. So in practice publications
in the field of term rewriting always pair a theoretical target: the domain of the
third list, with general rewriting or with some form of extended and/or restricted
rewriting: the domains of the first and/or the second list. Note that it is very well
possible to combine two extensions of the first list: conditional typed term rewriting
is a very useful combination in practice, or to combine extensions of the first list
with restrictions of the second list; there is nothing against typed orthogonal rewrite
systems. Most publications in the term rewriting area fit somewhere in this scheme.
Examples: a publication on the termination of typed term rewriting is for instance
[65], a publication on modular properties of conditional term rewriting systems is [46], and a publication on modular properties of constructor systems is [47]. The purpose of this analysis of the field, which may suffer from flaws or incompleteness, is merely to give an indication of the position and the context of the research which is documented in this thesis.

This dissertation is about a well-known variant of the basic rewrite principle: typed term rewriting, and a new theoretical direction: simulation and transformation techniques. Target of the research is to enlarge the practical applicability of term rewriting techniques. The approach in this thesis is to develop theory for the simulation and transformation of rewrite systems. It appears that term rewriting systems can often be transformed into systems which simulate the original system and which are simpler, or otherwise more desirable than the original system. Transformations can optimize criteria such as: sizes of terms, number of rewrite rules, number of rewrite steps, number of different operators in the signature, etc. It is also possible to acquire various properties which are desirable in certain practical circumstances by means of transformations. Work hypothesis in this thesis is:

The suitability of term rewriting systems to formalize subtree replacement techniques is enhanced if it is possible to transform term rewriting systems in systems which are simpler than the original system. A term rewriting system \( TRS_B \) is simpler than a term rewriting system \( TRS_A \) if:

- \( TRS_B \) has less rewrite rules than \( TRS_A \) or:
- the sizes of some (or all) terms in the term domain of \( TRS_B \) are less than the sizes of the terms in the term domain of \( TRS_A \) or:
- the number of different operators involved in \( TRS_B \) is less than the number of different operators involved in \( TRS_A \) or:
- \( TRS_B \) is a special case of \( TRS_A \) with regard to a certain criterion.

The notions term rewriting system, rewrite rules, size of terms, term domains, and operators, are supposed to have an intuitive meaning. They will be defined formally in the sequel. The last item of the work hypothesis is a general formulation which covers the earlier mentioned "various properties which are desirable in certain practical circumstances". Instances of this item are:

- a signature which contains no overloaded operators, instead of a general signature in which overloaded operators are allowed,
- a typing system (signature) which is many sorted instead of order sorted (many sorted typing is a special case of order sorted typing),
- a signature in which the maximal arity of the operators is 2, instead of a general signature in which any natural number arity is allowed,
• a rule set which is modularly decomposable.

The research question that will be answered in this thesis is:

The interest is in a general formalism that enables us to simulate typed term rewriting systems with rewrite systems which are in some sense simpler than the original rewrite system. What is an appropriate formalism for the simulation and transformation of typed term rewriting systems?

A formalism with the required properties will be presented and the simplifying capabilities of this formalism will be demonstrated by means of formal proofs, for each of the items of the work hypothesis. The practical value of some of the simplifications will be illustrated with some practical cases. It will be discussed what the practical value of the simplifications is and in what circumstances the practical benefit is most pronounced.

The structuring of this thesis is as follows. In chapter 2 the basics of signatures, terms, substitutions, rewrite relations, and the rudiments of algebraic semantics of terms, will be introduced. In chapter 3 three hierarchically related categories of simulation techniques for term rewriting systems will be presented. Properties of these simulation techniques will be stated in a formal way. The rest of the thesis is about particular instances of the generally defined simulation techniques. In chapter 4 two different simulation techniques will be deployed to eliminate multiply declared operators from an order sorted system. The significance of this elimination lies in that the simulations that will be introduced in chapter 6 require order sorted systems without multiply declared operators. A large example will be presented in chapter 4 to illustrate both alternatives for the elimination of multiply declared operators. In chapter 5 an alternative basic formalism for the construction of terms is presented. The advantage (and in fact the necessity) of this formalism over the conventional order sorted signature formalism will be discussed. Derivation schemes for the conversion of order sorted signatures to the new formalism and vice versa, and also for the conversion of context free grammars to the new formalism and vice versa, are given. In chapter 6 eight particular categories of simulations are presented. It is described how to transform a rewrite system into a simulating rewriting system for six so called identity transformations and for two so called structure transformations. The eight categories of transformations can be viewed as elementary and more complex transformations can be build by composition. The semantical aspects of the eight elementary transformations are subject of chapter 7. The extent of the elementary transformations of chapter 6, the space that is generated by these transformations and the composition primitive, will be addressed in chapter 8. In chapter 9 ways of combining the transformations of the preceding chapters in order to achieve certain simplifications, will be given. Chapter 10 consists of a practical case in which some aspects of the usefulness of the various transformations are illustrated. Chapter 11 concludes with a summary of directions for further research and with some final remarks. The contents of chapter 5 and 6 were published earlier (in a slightly
different form) in [63].

The dependencies of chapters is visualized as a building in which the bricks represent chapters.

![Diagram of dependencies of chapters]

*figure 1.1: dependencies of chapters*

The newly developed theory is stated almost entirely in standard first order predicate logic [19]. The following can be said about the notation of values, special values, and functions:

- The symbol $\bot$ denotes the special value *undefined*.

- *Tuples* are assumed to be sequences of values which can be elementary values or tuples. Tuples are denoted by placing the elements, comma separated, between angle brackets, e.g. $\langle e_1, \ldots, e_n \rangle$ is an $n$-tuple containing the elements $e_1, \ldots, e_n$. One-tuples are identified with their one element.

- *Relations* are denoted by sets of pairs (2-tuples). The binary infix notation and set-tuple notation are considered to be fully interchangeable, so $\langle e_1, e_2 \rangle \in R \iff e_1Re_2$.

- *Strings* are different from tuples. Strings are sequences of objects of the same sort which can be constructed out of smaller strings or out of elements by
means of a concatenation operation. The empty string is denoted by $\varepsilon$, the set of strings of length $n$ over symbol set $S$ is denoted by $S^n$, string concatenation (an associative operation) is denoted by ",", and the set of all finite length strings over symbol set $S$ is denoted by $S^*$. $S^+$ is defined to be the set of all finite, non-zero length strings over symbol set $S$. The length of an arbitrary string $s \in S^*$ is denoted by $|S|$; definition: $s \in S^n \Leftrightarrow |S| = n$. Like tuples, strings consisting of one element are identified with their one element. Two alternative notations are used for strings: ordinary juxtaposition of constituting elements and a notation with enclosing square brackets and comma separated elements. So the string $s$ consisting of the three elements $e_1, e_2, e_3$ can be denoted by $e_1e_2e_3$ and by $[e_1, e_2, e_3]$. The latter notation will be used in situations where the bounds of the string or of the constituting elements can be ambiguous. It is possible to address single elements from strings or to specify substrings of strings with the following notation. If $s = e_1...e_n$ then:

- $s(i) = e_i$,
- $s(i..j) = e_i...e_j$,
- $s(\$) = e_n$,
- $s(i..\$) = e_i...e_n$.

• About set comprehension and its notation: $\{e(x_1,...,x_n) \mid P(x_1,...,x_n)\}$ denotes the largest subset $V_s \subseteq V_r$, $V_r$ subset of a not further to be specified universe $U$, such that: $\forall x_1,...,x_n \in U [P(x_1,...,x_n) \Rightarrow e(x_1,...,x_n) \in V_s]$. If $V_r = U$ then the specification of $V_s$ will sometimes be omitted. The universe $U$ is assumed to be implicitly available in the context.

• The symbol $\mathcal{N}$ denotes the set of natural numbers $\{0, 1, 2, ...\}$.

• Function application is denoted by placing the argument, between parentheses, after the function identifier, e.g. $f(a)$ denotes the value of function $f$ applied to argument $a$. Functions can have only one argument. If more than one argument has to be supplied, the arguments have to be tupled. A convenient shorthand is the possibility to omit the brackets of a tuple when used in a function application, i.e. $f((a_1,...,a_n))$ is equivalent with $f(a_1,...,a_n)$.

• Function typing, denoted by $f : \text{type}_1 \rightarrow \text{type}_2$, has the following loose meaning: types are identified with sets and for a function $f$ with type $\text{type}_1 \rightarrow \text{type}_2$ it holds that:

- $\forall x \notin \text{type}_1 \{f(x) = \bot\}$,
- $\forall x \in \text{type}_1 \{f(x) = \bot \lor f(x) \in \text{type}_2\}$,

so $f : \text{type}_1 \rightarrow \text{type}_2$ is in general a partial function from $\text{type}_1$ to $\text{type}_2$. This perception of function typing implies that functions can have more than one type, but only one unambiguously defined least type.
• The construction \(\{a_1 \leftarrow v_1, \ldots, a_n \leftarrow v_n\}\) denotes the function that maps \(a_1\) to \(v_1\), \(\ldots\), \(a_n\) to \(v_n\) and all other argument values to \(\perp\). Type: \(\{a_1, \ldots, a_n\} \rightarrow \{v_1, \ldots, v_n\}\).

• The \textit{conditional expression} \(if \ P \ then \ T \ else \ E\) has its usual meaning, provided that the type of \(P\) is boolean and that the types of \(T\) and \(E\) are equal. Nested conditional expressions in the else-clause can be abbreviated with the well-known \textit{elif}-construction [64].

• The construction \(f \uparrow g\) denotes the function that maps argument value \(a\) to \(if \ g(a) = \perp \ then \ \perp \ else \ f(a) \ else \ g(a)\). The symbol \(\perp\) is a specially dedicated symbol for shrinking domains of functions.

• Each function can be extended from arguments to argument sets in the following way: \(f(A) = \{f(a) \mid a \in A\}\). Type: \(\mathcal{P}(\text{type}_1) \rightarrow \mathcal{P}(\text{type}_2)\) if \(f : \text{type}_1 \rightarrow \text{type}_2\). A set valued argument indicates that the related extension is meant by the function identifier, with \(\mathcal{P}(V)\) the powerset of the set \(V\).

• \textit{Functional composition} is denoted by the infix operator \(\circ\) and \(g \circ f\) is the function defined by: \((g \circ f)(x) = if \ f(x) = \perp \ then \ \perp \ else \ g(f(x))\). Type: \(\text{type}_1 \rightarrow \text{type}_3\) if \(f : \text{type}_1 \rightarrow \text{type}_2\) and \(g : \text{type}_2 \rightarrow \text{type}_3\).

• \textit{Precedence} of binary operators in predicate logic and other expressions is specified as follows (in descending order, operators on one and the same line have equal priority):

\[
\begin{align*}
\&,
\ast, /,
+,-,
=, \neq, <, >, \leq, \geq,
\cap, \cup, \setminus,
\subseteq, \supseteq, \subset, \supset,
\in, 
\wedge, 
\lor, 
\iff, 
\implies,
\leftarrow,
|.
\end{align*}
\]

Unary operators have always higher priority than binary operators (note that the minus operator is overloaded: it is used for subtraction and set difference).
• Binary operators with equal priority associate to the left: \( A - B \cup C \cap D = ((A - B) \cup C) \cap D \).

• Throughout the document sets will be denoted with capitals and functions with small Greek letters. Functions which are meant as operators will be denoted with three through six letter mnemonics, starting with a capital. Further notational conventions for notions to be introduced will be defined when appropriate.
Chapter 2

Basic Notions

2.1 Introduction

An introduction in the basics of signatures, terms, substitutions, rewrite relations, and the rudiments of algebraic semantics will be given in this chapter. Formal definitions are interleaved with theorems and comments to clarify things when necessary. This chapter is mainly to fix notation for notions which are well-known in the field of term rewriting. Most notions and notations that will be introduced in this chapter are taken from standard literature in the field of term rewriting (sometimes with slight modifications) [16, 41].

2.2 The Term Concept

Definition 2.1. An order sorted signature \( \Sigma \) is a quadruple \((S, Rdecs, Odecs, Vdecs)\) with:

- \( S \): a set of sorts or types,
- \( Rdecs \): a set of sub-sort declarations, each consisting of a pair \( \langle s_1, s_2 \rangle \) containing the two types \( s_1, s_2 \in S \),
- \( Odecs \): a set of operator declarations, each consisting of a pair \( \langle op, type-spec \rangle \) containing the operator \( op \) and the type specification \( type-spec \), of which \( type-spec \) in its turn a pair \( \langle s_1...s_n, s_r \rangle \) containing the operand type string \( s_1...s_n, s_r \in S^* \) and the result type \( s_r \in S \),
- \( Vdecs \): a set of variable declarations, each consisting of a pair \( \langle var, s \rangle \) containing the variable \( var \) and the type \( s \in S \),

for which the following hold:

1. the transitive closure of \( Rdecs \), when viewed as relation, is an irreflexive relation over \( S \),
2. the operand type strings of declarations of one and the same \( \text{op} \) all have one and the same length, implying that the arity of each \( \text{op} \) is determined unambiguously,

3. there is at most one declaration for each variable,

4. there are countably infinite variables for each type,

5. for each pair of (multiple) declarations \( \langle \text{op}, \langle s_1...s_n, s_r \rangle \rangle, \langle \text{op}, \langle s'_1...s'_n, s'_r \rangle \rangle \):
   
   5.1. \( s_1...s_n \neq s'_1...s'_n \),
   
   5.2. \( s_1...s_n < s'_1...s'_n \Rightarrow s_r \leq s'_r \),

   with \(<\) as defined below,

6. each type \( s \in S \) is inhabited, meaning that there are well-formed ground terms (definition 2.9) of that type.

Derived quantities that will be used frequently:

- \( \alpha = \{ \text{op} \leftarrow n \mid \langle \text{op}, \langle s_1...s_n, s_r \rangle \rangle \in Odecs \} \): the arity function, which returns for each operator its unique arity,

- \( \ll \): a partial order relation on \( S \), which is defined as the transitive closure of \( Rdecs \), denoted also by \( Rdecs^+ \); the symbol \( \leq \) denotes the reflexive closure of \( \ll \); the relation between types is extended to equal-length strings of types in the following way:

   - \( s_1...s_n = s'_1...s'_n \Leftrightarrow \forall 1 \leq i \leq n \ [s_i = s'_i] \),
   
   - \( s_1...s_n \leq s'_1...s'_n \Leftrightarrow \forall 1 \leq i \leq n \ [s_i \leq s'_i] \),
   
   - \( s_1...s_n < s'_1...s'_n \Leftrightarrow s_1...s_n \leq s'_1...s'_n \land s_1...s_n \neq s'_1...s'_n \),

- \( O = \{ \text{op} \mid \langle \text{op}, \text{type-spec} \rangle \in Odecs \} \): the set of operators,

- \( V = \{ \text{var} \mid \langle \text{var}, s \rangle \in Vdecs \} \): the set of variables,

- \( K = \{ \text{op} \in O \mid \alpha(\text{op}) = 0 \} \): the set of constants.

Components and associated derivable quantities of signatures will be indexed with a signature specifier whenever ambiguity can arise, so \( S_\Sigma \) and \( \alpha_\Sigma \) pertain to signature \( \Sigma \). An operator is said to be declared in a signature \( \Sigma \) if and only if it occurs at least once as first component of an operator declaration of \( Odecs \). An operator is said to be multiply declared if and only if it occurs at least twice as first component of an operator declaration of \( Odecs \). A signature is said to be plain if and only if it has no multiply declared operators. A signature is said to be finite if and only if \( S \) and \( Odecs \) are finite sets. In the rest of this thesis order sorted signatures are assumed to be finite.
2.2. **THE TERM CONCEPT**

Because of the practical background of the research we have chosen a definition of order sorted signatures which is very close to some concrete specification formalisms and languages [30, 6]. This meant for instance a primitive `Rdecs` with which it is possible to define a partial order relation `<` instead of extensionally specifying this relation. This has the advantage that pairs which are implied by the transitivity axiom of order relations do not need to be specified explicitly.

A least subtype declaration can be defined in terms of the partial order relation as follows (the minus superscript notation is based on the fact that the intended operation is the inverse of taking a reflexive closure): \( R^- = \{ (s_1, s_3) \in R | \neg \exists s_2 \in S \ [ (s_1, s_2) \in R \land (s_2, s_3) \in R] \} \). For order sorted signatures: \( Rdecs \supseteq \leq^{-} \).

The requirement of infinite variables for each type in order sorted signatures is not like concrete specification formalisms. The requirement is only to ensure that there are enough variables for each type. With finite sized terms it is always sufficient to have a certain finite number of variables (term sizes are in general unbounded but finite).

**Definition 2.2.** Let \( \Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle \) be a plain order sorted signature. The function \( Rtype : O_\Sigma \rightarrow S \) returns the result type of a given operator and the function \( Otype : O_\Sigma \times N \rightarrow S \) returns the operand type of a given operator at a given operand position. Formally:

- \( Rtype(op) = s_r \iff \langle op, \langle s_1 ... s_n, s_r \rangle \rangle \in Odecs \),
- \( Otype(op, i) = s_i \iff \langle op, \langle s_1 ... s_i ... s_n, s_r \rangle \rangle \in Odecs \).

Note that \( Rtype \) and \( Otype \) are well-defined if and only if \( \Sigma \) is plain. \( \square \)

**Definition 2.3.** Let \( \Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle \) be a plain order sorted signature. The function \( Fits : S \rightarrow \mathcal{P}(O_\Sigma) \) returns the set of operators which have result type less than or equal to the given type. It is the set of operators that "fits" at operand positions with the given type. Formally:

\[
Fits(s) = \{ op \in O_\Sigma | Rtype(op) \leq s \}.
\]

Note that \( Fits \) is well-defined if and only if \( \Sigma \) is plain. \( \square \)

**Definition 2.4.** A many sorted signature \( \Sigma \) is an order sorted signature with an empty relation `<` on the set of types \( S \), i.e. a signature in which for no pair of different types \( s_1, s_2 \in S : s_1 < s_2 \).

**Definition 2.5.** The set \( T \) of (general) terms over signature \( \Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle \) is defined inductively as the smallest set such that:

- \( \forall (var, s_v) \in Vdecs \ [ var \in T ] \),
- \( \forall (k, \langle e, s_r \rangle) \in Odecs \ [ k \in T ] \),
• \( \forall (\text{op}, (s_1 \ldots s_n, s_r)) \in \text{Odecs}, t_1 \in T, \ldots, t_n \in T \ [\text{op}(t_1, \ldots, t_n) \in T]. \) \hfill \Box

In the literature constants are often not considered as special cases but included in the general case of operators. We do not follow this practice because what we define is not merely terms but also our notation for terms. And as far as this notation is concerned constants form a special case: they are denoted without parentheses.

**Definition 2.6.** The set \( TG \) of ground terms over signature \( \Sigma = \langle S, \text{Rdecs}, \text{Odecs}, \text{Vdecs} \rangle \) is defined inductively as the smallest set such that:

• \( \forall (k, (e, s_r)) \in \text{Odecs} \ [k \in TG], \)

• \( \forall (\text{op}, (s_1 \ldots s_n, s_r)) \in \text{Odecs}, t_1 \in TG, \ldots, t_n \in TG \ [\text{op}(t_1, \ldots, t_n) \in TG]. \) \hfill \Box

**Definition 2.7.** The set \( T(s) \) of well-formed terms of type \( s \) over signature \( \Sigma = \langle S, \text{Rdecs}, \text{Odecs}, \text{Vdecs} \rangle \) is defined inductively as the smallest set such that:

• \( \forall (\text{var}, s_u) \in \text{Vdecs} \ [s_u \leq s \Rightarrow \text{var} \in T(s)], \)

• \( \forall (k, (e, s_r)) \in \text{Odecs} \ [s_r \leq s \Rightarrow k \in T(s)], \)

• \( \forall (\text{op}, (s_1 \ldots s_n, s_r)) \in \text{Odecs}, t_1 \in T(s_1), \ldots, t_n \in T(s_n) \)

\[ s_r \leq s \Rightarrow \text{op}(t_1, \ldots, t_n) \in T(s). \]

As an aid to memory for the difference between the definition of \( T \) and of \( T \), remember that the symbol \( T \) is more well-formed than the symbol \( T \). Note that what we call “well-formed terms” is in the literature sometimes referred to as well-typed terms. \hfill \Box

**Definition 2.8.** The set \( T \) of well-formed terms over signature \( \Sigma = \langle S, \text{Rdecs}, \text{Odecs}, \text{Vdecs} \rangle \) is defined as:

\[ T = \bigcup_{s \in S} T(s). \] \hfill \Box

**Definition 2.9.** The set \( TG \) of well-formed ground terms over signature \( \Sigma = \langle S, \text{Rdecs}, \text{Odecs}, \text{Vdecs} \rangle \) is defined as:

\[ TG = TG \cap T. \] \hfill \Box

The notations for term sets \( T, TG, T, TG \), will be indexed with signature specifiers whenever ambiguity can arise, so \( T_\Sigma, TG_\Sigma, T_\Sigma, TG_\Sigma \), are sets of terms over signature \( \Sigma \). Because \( T \supseteq TG, T \supseteq T, T \supseteq TG \), functions which are defined for \( T \) are also applicable to \( TG, T, TG \), this is in particular true for the functions \( \text{Var}, \text{Noc}, \text{Pos} \) and the functions for position arithmetic, denoted with the operators \( .[. \ldots \ldots ] \), to be introduced in the rest of this section.
Definition 2.10. An order sorted signature \( \Sigma = (S, Rdecs, Odecs, Vdecs) \) is called regular if and only if for any operand type string \( s_1...s_n \in S^* \) such that there is a \( \langle op, (s'_1,...,s'_n,s'_1) \rangle \in Odecs \) with \( s_1...s_n \leq s'_1...s'_n \) there is a unique least \( s''_1...s''_n \in S^* \) with \( s_1...s_n \leq s''_1...s''_n \) such that \( \langle op, (s''_1,...,s''_n,s'_1) \rangle \in Odecs. \)

Definition 2.11. An order sorted signature \( \Sigma = (S, Rdecs, Odecs, Vdecs) \) is called pre-regular if and only if for each \( op \in O \) and each type string \( s_1...s_n \in S^* \) the set \( \{s_r | \langle op, (s'_1,...,s'_n,s'_1) \rangle \in Odecs, s_1...s_n < s'_1...s'_n \} \) is either empty or has a unique least element.

Theorem 2.12. Regularity implies pre-regularity. \( \text{Proof:} \) suppose \( \Sigma = (S, Rdecs, Odecs, Vdecs) \) is not pre-regular, i.e. there is a specific \( op \) and \( s_1...s_n \in S^n \) such that the set \( \{s_r | \langle op, (s'_1,...,s'_n,s'_1) \rangle \in Odecs, s_1...s_n < s'_1...s'_n \} \) is non-empty and contains two different elements \( s_{r_1} \) and \( s_{r_2} \) which are both minimal. This means that there is no \( s''_n \) in the set such that \( s''_n < s_{r_1} \) or \( s''_n < s_{r_2} \). Let \( \langle op, (s'_1,...,s'_n,s'_1) \rangle \) and \( \langle op, (s''_1,...,s''_n,s''_1) \rangle \) be the corresponding declarations. By definition \( s_1...s_n < s''_1...s''_n \) and \( s_1...s_n < s'_1...s'_n \). In a regular signature: \( s''_1...s''_n < s'_1...s'_n \). Constraint 5.2 in definition 2.1 now implies that \( s_{r_1} < s_{r_2} \) or \( s_{r_2} < s_{r_1} \), a contradiction.

Theorem 2.13. In a pre-regular signature each term has one unambiguously defined least type. Formally: If \( \Sigma = (S, Rdecs, Odecs, Vdecs) \) is a pre-regular signature then:
\[
\forall t \in T, s_1, s_2 \in S [t \in T(s_1) \land t \in T(s_2) \Rightarrow s_1 \leq s_2 \lor s_2 \leq s_1].
\]

Proof with induction on the structure of terms. The proposition clearly holds for variables and constants. Now suppose \( t = op(t_1,...,t_n) \), with \( s_1,...,s_n \) the least types of \( t_1,...,t_n \), which are unique by the induction hypothesis. Pre-regularity means that the set \( \{s_r | \langle op, (s'_1,...,s'_n,s'_1) \rangle \in Odecs, s_1...s_n < s'_1...s'_n \} \) has a unique least element (it is not empty because \( t \in T \)). Therefore \( op(t_1,...,t_n) \) also has an unambiguously defined least type, which we had to prove.

2.3 Term Analysis

Some functions for the analysis and manipulation of terms will be given. Also the concept of rewrite system, which is a signature together with a set of rewrite rules, is defined formally in this section.

Definition 2.14. Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The function \( \text{Type} : T \rightarrow S \) returns the set of minimal types of a given term. Formally: \( \text{Type}(t) = \{s \in S | t \in T(s), \exists s' \in S [s' < s \land t \in T(s')]\} \). If it is obvious that the result of \( \text{Type} \) is a singleton set, which is the case for variables and constants, and for (general) terms over pre-regular signatures, it will sometimes be treated as a single element.

Definition 2.15. Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The function \( \text{Oper} : T \rightarrow \mathcal{P}(O) \) returns the set of operators that occur in a given term. A recursive definition:
\[ \text{Var}(v) = \emptyset, \text{if } v \in V; \]
\[ \text{Var}(k) = \{k\}, \text{if } k \in K; \]
\[ \text{Var}(\text{op}(t_1, \ldots, t_{\alpha(\text{op})})) = \{\text{op}\} \cup \bigcup_{i=1}^{\alpha(\text{op})} \text{Oper}(t_i), \text{if } \text{op} \in O - K. \]

**Definition 2.16.** Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The function \( \text{Var} : T \rightarrow \mathcal{P}(V) \) returns the set of variables that occur in a given term. A recursive definition:

- \( \text{Var}(v) = \{v\}, \text{if } v \in V; \)
- \( \text{Var}(k) = \emptyset, \text{if } k \in K; \)
- \( \text{Var}(\text{op}(t_1, \ldots, t_{\alpha(\text{op})})) = \bigcup_{i=1}^{\alpha(\text{op})} \text{Var}(t_i), \text{if } \text{op} \in O - K. \)

**Definition 2.17.** Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The function \( \text{Noc} : (O \cup V) \times T \rightarrow \mathcal{N} \) returns the number of occurrences of a given symbol in a given term. A recursive definition:

- \( \text{Noc}(\text{sym}, t) = 0, \text{if } t \in (K \cup V) \text{ and } \text{sym} \neq t; \)
- \( \text{Noc}(\text{sym}, t) = 1, \text{if } t \in (K \cup V) \text{ and } \text{sym} = t; \)
- \( \text{Noc}(\text{sym}, \text{op}(t_1, \ldots, t_{\alpha(\text{op})})) = \sum_{i=1}^{\alpha(\text{op})} \text{Noc}(\text{sym}, t_i), \)
  \hspace{1cm} \text{if } \text{op} \in (O - K) \text{ and } \text{sym} \neq \text{op};
- \( \text{Noc}(\text{sym}, \text{op}(t_1, \ldots, t_{\alpha(\text{op})})) = 1 + \sum_{i=1}^{\alpha(\text{op})} \text{Noc}(\text{sym}, t_i), \)
  \hspace{1cm} \text{if } \text{op} \in (O - K) \text{ and } \text{sym} = \text{op}. \)

**Definition 2.18.** Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). A term \( t \in T_\Sigma \) is called *linear* if and only if each variable of \( t \) occurs only once in \( t \). Formally:

- \( t \text{ is linear} \iff \forall v \in \text{Var}(t) \lnot [\text{Noc}(v, t) = 1]. \)

**Definition 2.19.** Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). A *substitution* is a total function \( \sigma : V \rightarrow T \), which is non-type-increasing, i.e. \( \forall v \in V \lnot [\sigma(v) \in T(\text{Type}(v))] \). The domain of substitutions is extended from variables to terms, \( \sigma : T \rightarrow T \), in such a way that:

- \( v^\sigma = \sigma(v), \text{for all variables } v \in V; \)
- \( k^\sigma = k, \text{for all constants } k \in K; \)
- \( \text{op}(t_1, \ldots, t_{\alpha(\text{op})})^\sigma = \text{op}(t_1^\sigma, \ldots, t_{\alpha(\text{op})}^\sigma), \text{for all operators } \text{op} \in O - K. \)
Superscript notation will be used for the extended version. Any substitution $\alpha : V \rightarrow V$ is called a variable substitution.

**Definition 2.20.** An alpha conversion of a term $t$ is the result of a reversible, non-identical substitution operation on $t$, i.e. a substitution operation determined by an $\alpha$ such that $t^\alpha \neq t$ and such that there is a substitution $\alpha'$ with $(t^\alpha)^{\alpha'} = t$. An alpha conversion of a pair of terms $(l, r)$ (equations, rewrite rules), is the result of a reversible, non-identical substitution operation applied to both components, i.e. a substitution operation determined by an $\alpha$ such that $(l^\alpha, r^\alpha) \neq (l, r)$ and such that there is a substitution $\alpha'$ with $((l^\alpha)^{\alpha'}, (r^\alpha)^{\alpha'}) = (l, r)$. An alpha conversion of a pair $(l, \alpha)$ consisting of a term $l$ and a variable substitution $\alpha$ is any pair $(l', \alpha')$ such that $l^\alpha$ and $l'^{\alpha'}$ are alpha-converible into each other. For a set $S$ of terms or pairs of terms $[S]_{\alpha}$ denotes a largest subset $S_{\text{sub}}$ of $S$ such that $S_{\text{sub}}$ contains no terms or pairs which are alpha convertible into each other. Note that $[S]_{\alpha}$ is unique up to systematic renaming of variables.

**Definition 2.21.** Let $\Sigma = \langle S, \text{Rdecs}, \text{Odecs}, \text{Vdecs} \rangle$. A variable $v$ is called maximally typed in a term $t \in T$ if and only if there is no variable substitution $\alpha = \{v' \leftarrow v\}$ with $v' \in V$ and $\text{Type}(v) < \text{Type}(v')$ and a term $t' \in T$ such that $t = t^{\alpha}$.

**Definition 2.22.** An order sorted rewrite system is a pair $\langle \Sigma, RS \rangle$ consisting of an order sorted signature $\Sigma$ and a rewrite system $RS$ consisting of a set of rewrite rules. Each rewrite rule in turn is a pair $(l, r)$ consisting of a left hand side pattern $l$ and a right hand side pattern $r$ for which:

- $l, r \in T,$
- $l \not\in V,$
- $\text{Var}(r) \subseteq \text{Var}(l),$
- $\forall s \in S \ [l \in T(s) \Rightarrow r \in T(s)].$

A many sorted rewrite system is an order sorted rewrite system $\langle \Sigma, RS \rangle$ of which $\Sigma$ is a many sorted signature. Order sorted and many sorted rewrite systems will mostly be abbreviated to just "rewrite system" if no ambiguity can arise.

The notion of order sorted rewrite system as fixed in definition 2.22 closely resembles the usual definition of the generally known notion of order sorted algebraic specification [50, 20, 33]. The only difference is in fact the directedness of the rules. In algebraic specifications these rules are usually given as equations which can be used in both directions in rewrite applications. We assume that the equations are directed ("oriented"), which can be seen as the result of a successfully completed Knuth-Bendix procedure or one of its extensions/variants [38, 3, 21], or as the design decision of a rule programmer who had his own reasons for imposing a certain direction on a rule (the purpose of a rewrite system will not always be the decision
of an instance of the word problem). It should be noted that systems that support algebraic specifications sometimes use the word “equation”, while assuming that these equations are directed. This is amongst others the case in [33].

Definition 2.23. Let \( \langle \Sigma, RS \rangle \) be an order sorted rewrite system. A rewrite rule \( \langle l, r \rangle \in RS \) is called left linear only if \( l \) is linear, right linear only if \( r \) is linear, and linear only if both \( l \) and \( r \) are linear. The system \( \langle \Sigma, RS \rangle \) is called left linear, right linear, or linear only if each rule \( h \in RS \) is left linear, right linear, or linear. \( \Box \)

Definition 2.24. Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The set of positions in a term \( t \in T \), denoted by \( Pos : T \rightarrow N^* \), is defined formally as:

- \( Pos(v) = \{ \varepsilon \} \), for all \( v \in V \);
- \( Pos(k) = \{ \varepsilon \} \), for all \( k \in K \);
- \( Pos(op(t_1, ..., t_{\alpha(op)})) = \{ i.p | 1 \leq i \leq \alpha(op), p \in Pos(t_i) \} \), for all \( op \in O - K \).

A position \( p \in N^* \) is called a prefix of \( p' \in N^* \) if there is a string \( s \in N^* \) such that \( ps = p' \). If \( p \) is a prefix of \( p' \) and \( p \neq p' \) then \( p \) may also be called a proper prefix of \( p' \). \( \Box \)

Definition 2.25. Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The subterm at position \( p \in Pos(t) \) in a term \( t \in T \), denoted by \( t|_p \), is defined formally as:

- \( t|_{\varepsilon} = t \),
- \( op(t_1, ..., t_{\alpha(op)})(i.p) = t_i|_p \).

A pattern \( l \in T \) is said to match in \( t \in T \) if and only if there is a position \( p \in Pos(t) \) and a substitution \( \sigma : V \rightarrow T \) such that \( t|_p = l^\sigma \). \( \Box \)

Definition 2.26. Let \( \Sigma = (S, Rdecs, Odecs, Vdecs) \). The replacement in \( t \in T \) of a subterm at position \( p \in Pos(t) \) by a term \( t_{new} \in T \), denoted by \( t|_{t_{new}}^p \), is defined formally as:

- \( t|_{t_{new}}^\varepsilon = t_{new} \),
- \( op(t_1, ..., t_n)|_{t_{new}}^p = op(t_1, ..., t_i|_{t_{new}}^p, ..., t_n) \).

Definition 2.27. Let \( \langle \Sigma, RS \rangle \) be an order sorted rewrite system. The one step rewrite relation \( \rightarrow_{RS} \) induced by a rewrite system \( RS \) is defined as: \( t \rightarrow_{RS} t' \) if and only if there exists:

- a position \( p \in Pos(t) \),
- a rule \( \langle l, r \rangle \in RS \),
2.4 SEMANTICS OF TERMS AND SIGNATURES

• a substitution $\sigma : V \to T$,

such that: $t|_{\sigma} = t^\sigma$ and $t' = t[\sigma]_p$. The transitive reflexive closure of $\to_X$ is denoted by: $\to_X^*$, the reverse relation of $\to_X$ by: $\leftarrow_X$ and the transitive reflexive closure of $\leftarrow_X$ by: $\leftarrow_X^*$.

\[\square\]

2.4 Semantics of Terms and Signatures

An introduction in the rudiments of algebraic semantics of typed terms will be presented. Only the aspects which are necessary for our purpose will be defined formally. Our purpose is to show that the various transformations of rewrite systems that will be introduced in the course of the thesis are compatible with algebraic semantics. The definitions are, with slight modifications, taken from [24, 54, 23].

Definition 2.28. Let $\Sigma = (S, Rdecs, Odecs, Vdecs)$ be an order sorted signature. A $\Sigma$-algebra $A$ is a pair $(S^A, O^A)$ consisting of:

• a set of domains $S^A$ which consists of a unique domain $s^A$ for each type $s \in S$,

• a set of operations $O^A$ which consists of a unique operation $op^A$ for each operator $op \in O$,

such that:

1. $s^A$ is a non-empty set for all domains $s^A \in S^A$,

2. if $s_1 < s_2$ then $s_1^A \subseteq s_2^A$,

3. if $(op, \langle s_1, ..., s_r \rangle) \in Odecs$ then for all $v_1 \in s_1^A, ..., v_n \in s_n^A$:

   $op^A(v_1, ..., v_n) \in s_r^A$.

The universe $U^A$ of $\Sigma$-algebra $A$ is defined as the union of all domains in $S^A$:

$U^A = \bigcup_{s \in S} s^A$.

\[\square\]

Problems with empty domains are alleviated by simply requiring that $s^A$ is non-empty for all $s^A \in S^A$. Empty domains have little practical significance and are therefore not considered.

Definition 2.29. Let $\Sigma$ be an order sorted signature and let $A, A'$ be two $\Sigma$-algebras. A surjective function $\phi_{A'} : U^A' \to U^A$ is called a homomorphism from $A'$ to $A$ if and only if:

\[\forall (op, \langle s_1, ..., s_r \rangle) \in Odecs, \forall v_1 \in s_1^A, ..., v_n \in s_n^A\]

\[\phi_{A'}(op^A(v_1, ..., v_n)) = op^A(\phi_{A'}(v_1), ..., \phi_{A'}(v_n))].\]
If $\phi_{A',A}$ is bijective between $U^A$ and $U^A$ then it may be called an isomorphism between $A$ and $A'$.

Our definition of homomorphisms is slightly less general than usual [20]. We require the homomorphism to be surjective because surjective homomorphisms (sometimes called coverings [54]) are the most interesting ones. Any non-surjective homomorphism from $A'$ to $A$ corresponds to a surjective homomorphism from $A'$ to a sub-algebra of $A$ (an algebra derived from $A$ by restricting the $A$-domains corresponding to the $\Sigma$-types). Domain restrictions like:

$$\forall s \in S [\phi_{A',A}(s'^A) \subseteq s^A],$$

are implied by the earlier stated general homomorphism requirement if any value in $U^A'$ is function value of some operation $op^A$ or some constant $c^A$, a very reasonable requirement. By excluding pathological cases we have obtained an elegant version of algebraic semantics of signatures and terms which is easy to handle for our purposes.

It is easy to see that the constraint $s_1 \ldots s_n < s'_1 \ldots s'_n \Rightarrow s_r \leq s'_r$ for multiply declared operators (constraint 5.2 of definition 2.1) is a very natural constraint with regard to algebraic interpretations. According to the definition of $\Sigma$-algebras, multiply declared operators $op$ pertain to one specific operation $op^A$. Now $s_r^A$ is of course always at most the co-domain of the non-restricted domain.

The constraint $s_1 \ldots s_n \neq s'_1 \ldots s'_n$ for multiply declared operators (constraint 5.1 of definition 2.1) is to prohibit declarations that do not add new information to the signature. For the same reason two declarations $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$, $\langle op, \langle s'_1 \ldots s'_n, s'_r \rangle \rangle$ for one operator $op$ and $s_1 \ldots s_n < s'_1 \ldots s'_n$ and $s_r = s'_r$ is in fact superfluous, i.e. adds no new information to the signature. However these pairs of multiple declarations are sometimes necessary to make a signature regular (definition 2.10). It is possible to transform a regular signature into a pre-regular signature for which $s_1 \ldots s_n < s'_1 \ldots s'_n \Rightarrow s_r < s'_r$ for all pairs of multiple declarations. Formally:

**Theorem 2.30.** Let $\Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle$ be an order sorted signature.

1. If $\Sigma$ is regular then there is a pre-regular $\Sigma'$ with $T_\Sigma = T_{\Sigma'}$ and with the property that for each pair of multiple declarations $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$, $\langle op, \langle s'_1 \ldots s'_n, s'_r \rangle \rangle$: $s_1 \ldots s_n < s'_1 \ldots s'_n \Rightarrow s_r < s'_r$.

2. If $\Sigma$ is pre-regular then there is a regular $\Sigma'$ with $T_\Sigma = T_{\Sigma'}$.

**Proof of 1.** Any regular signature is pre-regular, so the focus has to be on the pairs of multiple declarations $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$, $\langle op, \langle s'_1 \ldots s'_n, s'_r \rangle \rangle$ for which $s_1 \ldots s_n < s'_1 \ldots s'_n$ and $s_r = s'_r$ (or $s_r > s'_r$, or $s_r$ and $s'_r$ not related, is already prohibited by definition 2.1). The strategy is simple: drop any such $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$ from the specification. It is easy to see that $T_\Sigma$ remains the same after each drop: the operation of the declaration $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$ can entirely be covered by the declaration $\langle op, \langle s'_1 \ldots s'_n, s'_r \rangle \rangle$. It is also easy to see that the specification remains pre-regular after each drop: the sets in definition 2.11 will not lose elements and will therefore keep
the property of being empty or having a unique least element. Finally, it is easy to see that only a finite number of "drops" are necessary to acquire the property $s_1...s_n < s'_1...s'_n \Rightarrow s_r < s'_r$.

Proof of 2. Let $\langle \text{op, } \langle s'_1...s'_n, s'_r \rangle \rangle$, $\langle \text{op, } \langle s''_1...s''_n, s''_r \rangle \rangle$ be two declarations in which $s'_1...s'_n$ and $s''_1...s''_n$ are both minimal and $s_1...s_n$ a maximal type string such that $s_1...s_n < s'_1...s'_n$ and $s_1...s_n < s''_1...s''_n$. If the signature is pre-regular but not regular there always exists such a pair of declarations. Pre-regularity guarantees that $s'_r \leq s''_r$ or $s''_r \leq s'_r$ (or both). Now add to the signature the declaration $\langle \text{op, } \langle s_1...s_n, s_r \rangle \rangle$ with $s_r = s'_r$ if $s'_r \leq s''_r$ and $s_r = s''_r$ otherwise. It is easy to see that $T_S$ remains the same after the addition of this declaration: the operation of the newly added declaration in the construction of a term is always covered by one of the two original declarations. It is also easy to see that the pre-regularity property is preserved after an addition: the sets in definition 2.11 will not gain new elements and will therefore keep the property of being empty or having a unique least element. This process of adding a declaration can be repeated one or more times for a signature which is pre-regular but not regular. Termination is guaranteed because $<$ is well-founded on $S_n$ if $<$ is well-founded on $S$, and $S$ was assumed to be a finite set. When the process terminates the resulting signature is regular, so regularity can be attained after adding a finite number of extra declarations for some operators.

Example 2.31. Specification OddEvenMult1 is pre-regular with the property $s_1...s_n < s'_1...s'_n \Rightarrow s_r < s'_r$ and specification OddEvenMult2 is the corresponding regular specification.

Specification OddEvenMult1
That is, odd, even, nat;
That is, odd < nat, even < nat;
That is, mult: nat × nat × nat,
That is, even × nat × even,
That is, nat × even × even,
That is, odd × odd × odd;

End OddEvenMult1

Specification OddEvenMult2
That is, odd, even, nat;
That is, odd < nat, even < nat;
That is, mult: nat × nat × nat,
That is, even × nat × even,
That is, nat × even × even,
That is, even × even × even,
That is, odd × odd × odd;

End OddEvenMult2
Chapter 3

The Simulation Concept

3.1 Introduction

In order to reach our research goal: simplification of rewrite systems, we want to mimic the operation of one order sorted rewrite system by a different, simpler order sorted rewrite system. In this chapter a general theory will be developed for various kinds of simulation techniques for rewrite systems. We will abstract from the structure of terms and rewrite rules in this chapter and develop a theory for abstract rewrite systems. In the last section of this chapter is demonstrated how to apply the theory for abstract rewrite systems to order sorted rewrite systems.

Definition 3.1. An abstract rewrite system is a pair $\langle S, R \rangle$ consisting of a finite or countably infinite set $S$ and a binary relation $R$ on $S$. Two elements $t_1, t_2 \in S$ can be rewrite related. It is said that $t_1$ rewrites to $t_2$ if and only if $(t_1, t_2) \in R$. Notation: $t_1 \xrightarrow{R} t_2$ or $t_1 R t_2$. The transitive reflexive closure of $R$ (and $\rightarrow_R$) is denoted by $R^*$ or $\xrightarrow{R}$ and the reverse relation of $R$ (and $\rightarrow_R$) is denoted by $R^{-1}$ or $\leftarrow_R$. \hfill $\Box$

Two important properties of rewrite systems are termination (also called strong normalization) [17, 12] and confluence [51, 22]. A formal definition of these notions in terms of abstract rewrite systems:

Definition 3.2. An abstract rewrite system $\langle S, R \rangle$ is said to be terminating if and only if there is no infinite sequence $t_1, t_2, t_3, \ldots \in S$ (not necessarily all different) with $t_1 \xrightarrow{R} t_2 \xrightarrow{R} t_3 \xrightarrow{R} \ldots \ldots$. If the set $S$ is obvious from the context we will also use the notion termination in connection with only the binary relation $R$. \hfill $\Box$

Definition 3.3. An abstract rewrite system $\langle S, R \rangle$ is said to be confluent if and only if for all $t_1, t_2, t_3 \in S$ with $t_2 \leftarrow_R t_1 \xrightarrow{R} t_3$ there is a $t_4 \in S$ such that $t_2 \xrightarrow{R} t_4 \leftarrow_R t_3$. If the set $S$ is obvious form the context we will also use the notion confluence in connection with only the binary relation $R$. \hfill $\Box$

Let in the rest of this chapter $\langle S, R \rangle$ and $\langle S', R' \rangle$ be two abstract rewrite systems. We are going to explore ways in which $\langle S', R' \rangle$ can simulate the behaviour of
$\langle S, R \rangle$. We will focus on simulations in which each element in the simulated (original) domain is represented by one or more elements in the simulating domain. Let $S$ be the simulated domain and $S'$ the simulating domain. Let $\phi$ be a total, surjective function from $S'$ to $S$: the domain mapping. This $\phi$ induces a partition on $S'$ with the following shape:

$$[t'] = \{ t'' \in S' \mid \phi(t'') = \phi(t') \}.$$  

The equivalence classes $[t']$ are sets of representatives of the elements $\phi(t') \in S$. The simulation process can graphically be depicted as follows:

```
\begin{align*}
\text{input}' & \quad \overset{R'}{\longrightarrow} \quad \text{output}' \\
\phi & \quad \downarrow \\
\text{input} & \quad \overset{R}{\longrightarrow} \quad \text{output}
\end{align*}
```

*figure 3.1: The simulation principle*

Rewrite processes induced by rewrite relation $R$ on input, resulting in output, can be simulated by rewrite processes induced by $R'$ on input', resulting in output'. The value of input' in the simulating domain can be determined by taking a $\phi$-original of input, the reason for which $\phi$ has to be surjective. The rewrite systems $R$ and $R'$ are related in a way which is described in the rest of this chapter and which fulfills the requirements that are stated in the following definition.

**Definition 3.4.** Let $\phi$ be a total, bijective function from $S'$ to $S$ and let $\Phi^{-1}$ be the generalized inverse of $\phi$, i.e. $\Phi^{-1}(x) = \{ x' \in S' \mid \phi(x') = x \}$. The pair $\langle (S', R'), \phi \rangle$ will be called a *step simulator* of $\langle S, R \rangle$ if and only if:

1. $\forall t_1, t_2 \in S$ $[t_1 \rightarrow^*_R t_2 \Rightarrow \forall t'_1 \in \Phi^{-1}(t_1) \exists t'_2 \in \Phi^{-1}(t_2) [t'_1 \rightarrow^*_R t'_2]]$,

2. $\forall t_1, t_2 \in S$ $[t_1 \rightarrow^*_R t_2 \Leftrightarrow \exists t'_1 \in \Phi^{-1}(t_1), t'_2 \in \Phi^{-1}(t_2) [t'_1 \rightarrow^*_R t'_2]].$  \hfill $\Box$

### 3.2 Simple Step Simulation

Step simulation is simple if each class contains only one element. This is the case if $\phi$ is bijective from $S'$ to $S$. The simulations with a bijective $\phi$ will be called *simple step simulations*. A formal definition:

**Definition 3.5.** Let $\phi$ be a total, bijective function from $S'$ to $S$. The pair $\langle (S', R'), \phi \rangle$ will be called a *simple step simulator* of $\langle S, R \rangle$ if and only if:

$$\forall t'_1, t'_2 \in S' [t'_1 \rightarrow_R t'_2 \Leftrightarrow \phi(t'_1) \rightarrow_R \phi(t'_2)].$$  \hfill $\Box$
3.3. CLASS STEP SIMULATION

Each rewrite step in the $S$ domain corresponds to exactly one rewrite step in the $S'$ domain on $\phi$-corresponding terms and vice versa. A formal presentation of some useful properties of simple step simulators:

**Theorem 3.6.** If $\langle (S', R'), \phi \rangle$ is a simple step simulator of $\langle S, R \rangle$ then the following holds:

- $\langle (S', R'), \phi \rangle$ is a step simulator of $\langle S, R \rangle$,
- $\langle S, R \rangle$ is confluent if and only if $\langle S', R' \rangle$ is confluent,
- $\langle S, R \rangle$ terminates (is strongly normalizing) if and only if $\langle S', R' \rangle$ terminates.

**Proof:** easy. □

**Theorem 3.7.** If $\langle (S'', R''), \phi_2 \rangle$ is a simple step simulator of $\langle S', R' \rangle$ and $\langle (S', R'), \phi_1 \rangle$ a simple step simulator of $\langle S, R \rangle$ then $\langle (S'', R''), \phi_1 \circ \phi_2 \rangle$ is a simple step simulator of $\langle S, R \rangle$. **Proof:** easy. □

Two generalizations of the simple step simulation notion will be introduced in the sequel. The generalizations are two alternative approaches for the case that $\phi$ is surjective but not bijective.

### 3.3 Class Step Simulation

If the classes of representatives are not singleton sets but sets containing possibly more than one element, the definition of simulation becomes more complicated. A maximal preserving of properties of rewrite processes is attained if the fact that $t_1$ can be rewritten into $t_2$ implies that each representative of $t_1$ can be rewritten into some representative of $t_2$ and the fact that $t_1$ cannot be rewritten into $t_2$ implies that none of the representatives of $t_1$ can be rewritten into any of the representatives of $t_2$. This kind of simulation will be called class step simulation. A formal definition:

**Definition 3.8.** Let $\phi$ be a total, surjective function from $S'$ to $S$ and let $\Phi^{-1}$ be the generalized inverse of $\phi$, i.e. $\Phi^{-1}(x) = \{x' \in S' \mid \phi(x') = x\}$. The pair $\langle (S', R'), \phi \rangle$ will be called a class step simulator of $\langle S, R \rangle$ if and only if:

$$\forall t_1, t_2 \in S \left[ t_1 \rightarrow_R t_2 \Rightarrow \forall t'_1 \in \Phi^{-1}(t_1) \exists t'_2 \in \Phi^{-1}(t_2) \left[ t'_1 \rightarrow_{R'} t'_2 \right] \right] \land$$

$$t_1 \rightarrow_R t_2 \iff \exists t'_1 \in \Phi^{-1}(t_1), t'_2 \in \Phi^{-1}(t_2) \left[ t'_1 \rightarrow_{R'} t'_2 \right] \]$$

□

There are only rewrite steps between representatives of $t_1$ and $t_2$ if the represented terms $t_1, t_2$ are rewrite related and in that case each representative of $t_1$ is rewrite related with at least one representative of $t_2$.

**Theorem 3.9.** If $\langle (S', R'), \phi \rangle$ is a class step simulator of $\langle S, R \rangle$ then the following holds:
• \( \langle S', R' \rangle, \phi \) is a step simulator of \( \langle S, R \rangle \),
• \( \langle S, R \rangle \) terminates \( \iff \langle S', R' \rangle \) terminates,
• \( \langle S, R \rangle \) is confluent \( \iff \)
\[
\forall t_1', t_2', t_3' \in S' \exists t_4', t_5' \in S' [t_1' \overset{R'}{\rightarrow} t_2' \land t_1' \overset{R'}{\rightarrow} t_3' \Rightarrow
t_2' \overset{R'}{\rightarrow} t_4' \land t_3' \overset{R'}{\rightarrow} t_5' \land \phi(t_4') = \phi(t_5')].
\]

Proof of step simulation: easy with induction on the length of the rewrite sequence.

Proof of termination: each rewrite step in the original system corresponds to exactly one rewrite step in the simulating system and vice versa so the equivalence of termination is evident.

Proof of confluence. The proof relies heavily on the fact that each composed step simulator is a step simulator and that:

I. \( \forall t_1, t_2 \in S [t_1 \overset{R}{\rightarrow} t_2 \Rightarrow \forall t_1' \in \Phi^{-1}(t_1) \exists t_2' \in \Phi^{-1}(t_2) [t_1' \overset{R'}{\rightarrow} t_2'] \],

II. \( \forall t_1, t_2 \in S [t_1 \overset{R}{\rightarrow} t_2 \Leftarrow \exists t_1' \in \Phi^{-1}(t_1), t_2' \in \Phi^{-1}(t_2) [t_1' \overset{R'}{\rightarrow} t_2'] \].

Proof of \( \Rightarrow \). Let \( t_1', t_2', t_3' \in S' \) with \( t_2' \overset{R'}{\leftarrow} t_1', t_3' \overset{R'}{\rightarrow} t_3 \). From II. and the totality of \( \phi \) follows that \( \phi(t_2') \overset{R}{\leftarrow} \Phi^{-1}(t_2'), \phi(t_3') \overset{R}{\rightarrow} \Phi^{-1}(t_3) \) and from the confluence of \( R \) that there is a \( t \in S \) such that \( \phi(t_2') \overset{R}{\rightarrow} t \overset{R}{\leftarrow} \Phi(t_3) \). From I. now follows that \( \exists t_4', t_5' \in \Phi^{-1}(t) [t_2' \overset{R}{\rightarrow} t_4', t_3' \overset{R}{\rightarrow} t_5' \]}. Obviously \( \phi(t_4') = \phi(t_5') \), which we had to prove.

Proof of \( \Leftarrow \). Let \( t_1, t_2, t_3 \in S \) with \( t_2 \overset{R}{\leftarrow} t_1 \overset{R}{\rightarrow} t_3 \). From I. follows that for all \( t_1' \in \Phi^{-1}(t_1) \) there are \( t_2' \in \Phi^{-1}(t_2), t_3' \in \Phi^{-1}(t_3) \) such that \( t_2' \overset{R}{\rightarrow} t_1' \overset{R}{\rightarrow} t_3' \). The premise now says that there are \( t_4', t_5' \) with \( \phi(t_4') = \phi(t_5') \) and \( t_2' \overset{R}{\rightarrow} t_4', t_3' \overset{R}{\rightarrow} t_5' \). From I and the totality of \( \phi \) follows that \( t_2 \overset{R}{\rightarrow} \Phi^{-1}(t) \phi(t_4') = \phi(t_5') \overset{R}{\leftarrow} t_3 \), hence \( R = R \) is confluent.

\( \square \)

**Theorem 3.10.** If \( \langle S'', R'' \rangle, \phi_2 \) is a class step simulator of \( \langle S', R' \rangle \) and \( \langle S', R' \rangle, \phi_1 \) a class step simulator of \( \langle S, R \rangle \) then \( \langle S'', R'' \rangle, \phi_1 \circ \phi_2 \) is a class step simulator of \( \langle S, R \rangle \).

**Proof:** let \( \phi = \phi_1 \circ \phi_2 \), and \( \Phi^{-1}, \Phi_1^{-1}, \Phi_2^{-1} \) the corresponding generalized inverses of \( \phi, \phi_1, \phi_2 \). It is easy to see that \( \phi \) is a total, surjective function. We have to prove two facts:

I. \( \forall t_1, t_2 \in S [t_1 \overset{R}{\rightarrow} t_2 \Rightarrow \forall t_1'' \in \Phi^{-1}(t_1) \exists t_2'' \in \Phi^{-1}(t_2) [t_1'' \overset{R}{\rightarrow} t_2''] \],

II. \( \forall t_1, t_2 \in S [t_1 \overset{R}{\rightarrow} t_2 \Leftarrow \exists t_1'' \in \Phi^{-1}(t_1), t_2'' \in \Phi^{-1}(t_2) [t_1'' \overset{R}{\rightarrow} t_2''] \].

I. Suppose \( t_1 \overset{R}{\rightarrow} t_2 \). For all \( t_1'' \in \Phi^{-1}(t_1) \) there is a \( t_1' \in \Phi_1^{-1}(t_1) \) such that \( t_1' \in \Phi_2^{-1}(t_4) \). Because \( \langle S', R' \rangle, \phi_1 \) is a class step simulator of \( \langle S, R \rangle \) there is for all \( t_1' \in \Phi_1^{-1}(t_1) \) a \( t_2' \in \Phi_1^{-1}(t_2) \) such that \( t_1' \overset{R}{\rightarrow} t_2' \). Because \( \langle S'', R'' \rangle, \phi_2 \) is a class step simulator of \( \langle S', R' \rangle \) there is for all \( t_2' \in \Phi_1^{-1}(t_2) \) a \( t_2'' \in \Phi_1^{-1}(t_2) \) such that \( t_2' \overset{R}{\rightarrow} t_2'' \). Therefore for all \( t_1'' \in \Phi^{-1}(t_1) \) there is a \( t_2'' \in \Phi^{-1}(t_2) \) such that \( t_1'' \overset{R}{\rightarrow} t_2'' \), hence \( \phi(t_1'') = \phi(t_2'') \).

\( \forall t_1'' \in \Phi^{-1}(t_1) \exists t_2'' \in \Phi^{-1}(t_2) [t_1'' \overset{R}{\rightarrow} t_2''] \].
3.4. COMPOSED STEP SIMULATION

simulator of \( \langle S', R' \rangle \) there is for all \( t_1' \in \Phi_2^{-1}(t_1') \) a \( t_2' \in \Phi_2^{-1}(t_2') \) such that \( t_1' \rightarrow_{R'} t_2' \). The fact to be proven now follows from \( \Phi_2^{-1}(t_2') \subseteq \Phi_1^{-1}(t_2) \) for all \( t_2' \in \Phi_1^{-1}(t_2) \).

II. Suppose \( t_1 \not\rightarrow_R t_2 \). Because \( \langle (S', R'), \phi_1 \rangle \) is a class step simulator of \( \langle S, R \rangle \), there is no \( t_1' \in \Phi_1^{-1}(t_1) \) and no \( t_2' \in \Phi_1^{-1}(t_2) \) with \( t_1' \rightarrow_{R'} t_2' \). Moreover, \( t_1'' \rightarrow_{R'} t_2'' \) with \( t_1'' \in \Phi_1^{-1}(t_1) \), \( t_2'' \in \Phi_1^{-1}(t_2) \) would imply that \( t_1'' \rightarrow_{R'} t_2'' \) for some \( t_1' \in \Phi_1^{-1}(t_1) \) and some \( t_2' \in \Phi_1^{-1}(t_2) \), a contradiction. Therefore \( t_1'' \not\rightarrow_{R'} t_2'' \). □

3.4 Composed Step Simulation

Class step simulation does not blend very well with order sorted rewrite systems with non-left-linear rules. For an important category of domain mappings it is not possible to derive a finite rule class step simulator from a finite rule rewrite system if one or more of the original rules are non-left-linear. The reason for this will be discussed in section 3.5.

This and other reasons make it in some situations undesirable or impossible to rewrite each representative of \( t_1 \) to some representative of \( t_2 \) if \( t_1 \) and \( t_2 \) are rewrite related in the original domain. A solution to this problem is: select one fixed element from each class of representatives and design the simulating system such that each element of the class of \( t_1 \) is rewrite related (via zero, one or more steps) with the particular selected element and this selected element rewrite related (via one step) with some element of the class of \( t_2 \) (if \( t_1 \) and \( t_2 \) are rewrite related in the original domain).

A problem with the just described simulation process is that the "main" rewrite steps, the rewrite steps between two classes, and the "auxiliary" rewrite steps, the rewrite steps within one class, coincide if the rewrite relation to be simulated is not irreflexive, i.e. if the domain to be simulated contains elements which rewrite to themselves in one step. Fortunately non-irreflexive rewrite systems are not very useful in practice and easily recognizable when specified with transformation rules on term sets. So there is not much lost if these systems are precluded from the scope of this sophisticated form of step simulation. If non-irreflexive systems are precluded it is easy to tell whether the simulating rewrite step is "main" or "auxiliary". When applied to term rewriting systems it will in practice often be the case that the categorization of rewrite steps induces a partition on the rule set, i.e. the function of each rule is fixed: performing only main steps or only auxiliary steps. This phenomenon can be utilized to verify the formal definition of this variant of step simulation, that will be called composed step simulation. The selected element in each class is determined by \( \psi \).

Definition 3.11. Let \( \phi \) be a total, surjective function from \( S' \) to \( S \) and let \( \Phi^{-1} \) be the generalized inverse of \( \phi \). Let \( \psi \) be a total, injective function from \( S \) to \( S' \) such that for all elements \( t \) in \( S' \): \( \phi(\psi(t)) = t \). The triple \( \langle (S', R'), \phi, \psi \rangle \) will be called a composed step simulator of \( \langle S, R \rangle \) if and only if:

- \( R \) has an irreflexive one step rewrite relation and
CHAPTER 3. THE SIMULATION CONCEPT

- \( \forall t' \in S' \ [t' \xrightarrow{\ast_R} \psi(\phi(t'))] \) and

- \( \forall t_1, t_2 \in S \ [t_1 \xrightarrow{R} t_2 \Rightarrow \exists t'_2 \in \Phi^{-1}(t_2) \ [\psi(t_1) \xrightarrow{R} t'_2] \wedge \]

\[ t_1 \xrightarrow{R} t_2 \iff \exists t'_1 \in \Phi^{-1}(t_1), t'_2 \in \Phi^{-1}(t_2) \ [t'_1 \xrightarrow{R} t'_2]. \] \( \square \)

**Theorem 3.12.** If \( \langle S', R' \rangle, \phi, \psi \) is a composed step simulator of \( \langle S, R \rangle \) then \( \langle S', R' \rangle, \phi \) is a step simulator of \( \langle S, R \rangle \).

**Proof:** easy with induction on the length of the rewrite sequences. \( \square \)

**Theorem 3.13.** Let \( \langle S', R' \rangle, \phi, \psi \) be a composed step simulator of \( \langle S, R \rangle \). \( \langle S', R' \rangle \) terminates if and only if:

- \( \langle S, R \rangle \) terminates and

- \( \langle S', R' \rangle \) admits no infinite rewrite sequences \( t'_1 \xrightarrow{R} t'_2 \xrightarrow{R} t'_3 \xrightarrow{R} \ldots \)

with \( \phi(t'_1) = \phi(t'_2) = \phi(t'_3) = \ldots \)

**Proof of \( \Rightarrow \).** From the foregoing it will be clear that each rewrite sequence of \( R' \) consists of a number of "main" rewrite steps, interleaved with zero, one, or more "auxiliary" rewrite steps. If the number of main rewrite steps is finite and the total sequence is non-terminating then there must be a non-terminating auxiliary sequence, contradicting the assumptions.

**Proof of \( \Leftarrow \).** It is easy to construct a non-terminating \( R' \) sequence if \( R \) doesn’t terminate, and trivial if \( R' \) doesn’t terminate in one of the classes of representatives. \( \square \)

**Theorem 3.14.** Let \( \langle S', R' \rangle, \phi, \psi \) be a composed step simulator of \( \langle S, R \rangle \). \( \langle S', R' \rangle \) is confluent if and only if \( \langle S, R \rangle \) is confluent.

**Proof.** The proof relies heavily on the fact that each composed step simulator is a step simulator and that:

I. \( \forall t_1, t_2 \in S \ [t_1 \xrightarrow{R} t_2 \Rightarrow \forall t'_1 \in \Phi^{-1}(t_1) \exists t'_2 \in \Phi^{-1}(t_2) \ [t'_1 \xrightarrow{R} t'_2]], \)

\( \forall t_1, t_2 \in S \ [t_1 \xrightarrow{R} t_2 \iff \exists t'_1 \in \Phi^{-1}(t_1), t'_2 \in \Phi^{-1}(t_2) \ [t'_1 \xrightarrow{R} t'_2]]. \)

**Proof of \( \Rightarrow \).** Let \( t'_1, t'_2, t'_3 \in S' \) with \( t'_2 \xleftarrow{R} t'_1 \xrightarrow{R} t'_3 \). From II. and the totality of \( \phi \) follows that \( \phi(t'_2) \xleftarrow{R} \phi(t'_1) \xrightarrow{R} \phi(t'_3) \) and from the confluence of \( \langle S, R \rangle \) that there is a \( t_4 \in S \) such that \( \phi(t'_2) \xrightarrow{R} t_4 \xleftarrow{R} \phi(t'_3) \). From I. now follows that \( \exists t'_4, t''_4 \in \Phi^{-1}(t_4) \ [t'_2 \xrightarrow{R} t'_4 \wedge t'_3 \xrightarrow{R} t''_4] \) and from definition 3.11 that \( t'_4 \xrightarrow{R} \psi(t_4) \xleftarrow{R} t'_3 \), hence \( \langle S', R' \rangle \) is confluent.

**Proof of \( \Leftarrow \).** Let \( t_1, t_2, t_3 \in S \) with \( t_2 \xleftarrow{R} t_1 \xrightarrow{R} t_3 \). From I. follows that for all \( t'_1 \in \Phi^{-1}(t_1) \) there are \( t'_2 \in \Phi^{-1}(t_2), t'_3 \in \Phi^{-1}(t_3) \) such that \( t'_2 \xleftarrow{R} t'_1 \xrightarrow{R} t'_3 \). From the confluence of \( \langle S', R' \rangle \) follows that there exists a \( t'_4 \in S' \) with \( t'_2 \xrightarrow{R} t'_4 \xleftarrow{R} t'_3 \).
and from II. that there exists a \( t_4 \in S \), namely \( t_4 = \phi(t'_4) \), such that \( t_2 \xrightarrow{R} t_4 \xleftarrow{R} t_3 \). Hence \( (S, R) \) is confluent. \( \square \)

**Theorem 3.15.** If \( \langle \langle S''', R''' \rangle, \phi_2, \psi_2 \rangle \) is a composed step simulator of \( \langle S', R' \rangle \) and \( \langle \langle S', R' \rangle, \phi_1, \psi_1 \rangle \) a composed step simulator of \( \langle S, R \rangle \) then \( \langle \langle S'', R'' \rangle, \phi_1 \circ \phi_2, \psi_2 \circ \psi_1 \rangle \) is a composed step simulator of \( \langle S, R \rangle \).

*Proof:* let \( \phi = \phi_1 \circ \phi_2 \), \( \psi = \psi_2 \circ \psi_1 \), and \( \Phi^{-1}, \Phi_1^{-1}, \Phi_2^{-1} \) the corresponding generalized inverses of \( \phi, \phi_1, \phi_2 \). It is easy to see that \( \phi \) is a total, surjective function and that \( \phi(\psi(t)) = t \) for all \( t \in S \). We have to prove three facts:

I. \( \forall t'' \in S'' \ [t'' \xrightarrow{R''} \psi(\phi(t''))] \)

II. \( \forall t_1, t_2 \in S \ [t_1 \rightarrow_R t_2 \iff \exists t'' \in \Phi^{-1}(t_2) \ [\psi(\phi(t'_1)) \rightarrow_{R'} t''] \] \)

III. \( \forall t_1, t_2 \in S \ [t_1 \rightarrow_R t_2 \iff \exists t'_1 \in \Phi^{-1}(t_1), t'' \in \Phi^{-1}(t_2) \ [t'_1 \rightarrow_{R'} t''] \] \)

Lemma: \( \forall t'_1, t'_2 \in S' \ [t'_1 \rightarrow_{R'} t'_2 \Rightarrow \forall t'_1 \in \Phi^{-1}(t'_1), t'' \in \Phi^{-1}(t'_2) \ [t'_1 \rightarrow_{R'} t''] \] (the first property of step simulators with a renaming of variables to make it fit better in the proof).

I. If \( t'' \in \Phi^{-1}(\psi(\phi(t''))) \) then \( \psi(\phi(t'')) = \psi(\phi(t'')) \) and \( t'' \xrightarrow{R''} \psi(\phi(t'')) \) because \( \langle \langle S'', R'' \rangle, \phi_2, \psi_2 \rangle \) is a composed step simulator of \( \langle S', R' \rangle \). If \( t'' \in \Phi^{-1}(t') \) for some \( t' \neq \psi(\phi(t'')) \) then \( t' \xrightarrow{R} \psi(\phi(t'')) \) because \( \langle \langle S', R' \rangle, \phi_1, \psi_1 \rangle \) is a composed step simulator of \( \langle S, R \rangle \) and there is (lemma) \( t'' \in \Phi^{-1}(\psi(\phi(t''))) \) such that \( t'' \xrightarrow{R'} t' \), \( \psi(\phi(t'')) \) is a composed step simulator of \( \langle S, R \rangle \). For reasons mentioned earlier: \( t'' \xrightarrow{R'} \psi(\phi(t'')) \), which means that: \( t'' \xrightarrow{R} \psi(\phi(t'')) \).

II. Suppose \( t_1 \rightarrow_R t_2 \). Because \( \langle \langle S', R' \rangle, \phi_1, \psi_1 \rangle \) is a composed step simulator of \( \langle S, R \rangle \), there is a \( t'_2 \in \Phi^{-1}(t_2) \) such that \( \psi_1(t_1) \rightarrow_R t'_2 \). Because \( \langle \langle S'', R'' \rangle, \phi_2, \psi_2 \rangle \) is a composed step simulator of \( \langle S, R \rangle \), there is a \( t'_2 \in \Phi^{-1}(t'_2) \) such that \( \phi_2(\psi(t'_1)) \rightarrow_R t'' \). The fact to be proven follows now from: \( \phi(\psi(t'_1)) = t'_1 \) and \( \Phi^{-1}(t'_2) \subseteq \Phi^{-1}(t_2) \).

III. Suppose \( t_1 \not\rightarrow_R t_2 \). Because \( \langle \langle S', R' \rangle, \phi_1, \psi_1 \rangle \) is a composed step simulator of \( \langle S, R \rangle \), there is no \( t'_1 \in \Phi^{-1}(t_1) \) and no \( t'_2 \in \Phi^{-1}(t_2) \) with \( t'_1 \rightarrow_R t'_2 \). Moreover, \( t'_1 \rightarrow_R t'' \) \( t'_2 \) \( t'' \in \Phi^{-1}(t_1), t'' \in \Phi^{-1}(t_2) \) would imply that \( t'_1 \rightarrow_R t'_2 \) for some \( t'_1 \in \Phi^{-1}(t_1) \) and some \( t'_2 \in \Phi^{-1}(t_2) \), a contradiction. Therefore \( t'_1 \not\rightarrow_R t'' \) \( t'_2 \).

### 3.5 Simulations of Term Rewriting Systems

Each order sorted rewrite system \( \langle \Sigma, R \rangle \) (definition 2.22) induces an abstract rewrite system with shape \( \langle T_\Sigma, \rightarrow_R \rangle \), so the general theory for simulation can be applied to order sorted term rewriting systems. In chapter 4 the general theory will be applied to order sorted term rewriting in order to eliminate overloading from order sorted signatures. In chapter 6 the general theory will be applied to a different basic formalism: the retracted vocabulary formalism which will be introduced in chapter 5. Retracted vocabularies are similar to order sorted signatures in that
they define a subset of an inductively defined set of general terms to be well-formed terms. The basics of term rewriting for retrenced vocabularies are the same as they are for order sorted signatures, so the general simulation theory is also applicable to retrenced vocabularies.

In fact any selection of well-formed terms from an inductively defined general term set can be used as a simulating domain for the simulation of an order sorted term rewrite system, provided that the simulating rewrite system is well-formed preserving on that selection. One useful way of restricting the inductively defined general set of terms is to preclude terms with variables. This subset of terms, commonly referred to as ground terms, is closed under rewriting due to the requirement that all variables in right hand side patterns of rules have to occur in left hand side patterns as well. The usefulness of restricting simulations to ground rewriting is that sometimes the conditions for the simulations are too strong to be satisfied for general rewriting.

The notion of simple ground step rewriting will be used so extensively in the rest of this thesis that a definition in terms of the term rewriting concept will be given explicitly:

**Definition 3.16.** Let $\langle \Sigma, RS \rangle$, $\langle \Sigma', RS' \rangle$ be two order sorted rewrite systems and let $\phi$ be a total, bijective function from $T\Sigma$ to $T\Sigma'$. The pair $\langle \langle \Sigma', RS' \rangle, \phi \rangle$ will be called a **simple ground step simulator** of $\langle \Sigma, RS \rangle$ if and only if:

$$\forall t_1', t_2' \in T\Sigma' \ [t_1' \rightarrow_{RS'} t_2' \Leftrightarrow \phi(t_1') \rightarrow_{RS \phi(t_2')}].$$

The transformation of a rewrite system is defined to be the derivation of a simulating rewrite system from a given rewrite system, which simulates the original system according to one of the simulation principles that were defined in this section, and with regard to given term domains and a given domain mapping between these domains.

In section 3.3 was stated that class step simulation did not blend very well with order sorted rewrite systems with non-left-linear rules. This is so because for an important category of domain mappings it is not possible to derive a finite rule class step simulator for a finite rule rewrite system if one or more of the rules are non-left-linear. One of the categories of domain mappings that pose problems is the category that has a property which is in other situations often called **stability** [16].

**Definition 3.17.** Let $\Sigma, \Sigma'$ be two signatures. A domain mapping $\phi : T\Sigma' \rightarrow T\Sigma$ is called **stable** if and only if:

$$\forall t_1', t_2' \in T\Sigma' \ [\phi(t_1') = \phi(t_2')] \Rightarrow \forall t' \in T\Sigma', p' \in Pos(t') \ [t'|_{p'} = t_1' \Rightarrow \phi(t') = \phi(t'[t_2'|_{p'})]].$$

Verbally: replacing sub-terms with $\phi$-equivalent sub-terms results in $\phi$-equivalent terms. The property can be expressed very conveniently with fragment notation, a
notation which will only be developed formally for the retrenched vocabulary formal-
ism: definition 5.34 and 5.37. A domain mapping \( \phi : \mathcal{T}_\Sigma \rightarrow \mathcal{T}_\Sigma \) is called stable if
and only if:

\[ \forall t'_1, t'_2 \in \mathcal{T}_\Sigma : [\phi(t'_1) = \phi(t'_2)] \Rightarrow \forall r \in \mathcal{F}\mathcal{T}_\Sigma : [\phi(fr[t'_1]) = \phi(fr[t'_2])]. \]

The domain mappings associated with our overloading resolution (chapter 4)
are all stable. It is in general not possible to cover the functionality of a non-left-
linear rule with a finite number of simulating rules in the case of a stable, non-bijective
domain mapping. This is because it is not possible to determine \( \phi \)-equivalence of
the sub-terms to be matched against a non-linear (i.e. multiple occurring) variable
with a finite number of patterns. An example to illustrate this.

**Example 3.18.** Matching of non-linear patterns in \( \phi \)-equivalent terms. Consider
two many sorted signatures \( \Sigma = \langle S, Odecs, Vdecs \rangle, \Sigma' = \langle S', Odecs', Vdecs' \rangle \) with:

\[ S = S' = \{ s, t \}, \]

\[ Odecs = \{ (f, (ss, t)), (g, (s, s)), (a, (s, s)) \}, \]

\[ Odecs' = \{ (f, (ss, t)), (g, (s, s)), (a_1, (s, s)), (a_2, (s, s)) \}, \]

\[ Vdecs = Vdecs' = \{ (x, s) \}, \]

and consider the surjective domain mapping \( \phi \) that maps each \( \Sigma' \)-term to a \( \Sigma \)-term
by removing the index from the constant \( a \). This \( \phi \) is easily be seen to be stable.
Now consider the following subset of \( \mathcal{T}G_\Sigma \):

\[ Ss = \{ f(a_1, a_2), f(g(a_1), g(a_2)), f(g(g(a_1)), g(g(a_2))), ... \}. \]

It is easy to see that the pattern \( f(x, x) \) matches in each \( \phi(t) \) with \( t \in Ss \) and that
there is no finite set of patterns \( SP \) such that:

\[ f(x, x) \text{ matches in } \phi(t) \Leftrightarrow \exists p \in SP [p \text{ matches in } t]. \]

This means that there are term domains, stable domain mappings between these
term domains and finite rule rewrite systems for one of these domains, for which
there is no finite rule class step simulator in the other domain. The composed step
simulation technique does not have this drawback.

**3.6 Composition of Different Simulation Techniques**

The main purpose for the introduction of the class step simulation and composed
step simulation technique is to enlarge the applicability of the various instances of
simple step simulation that will be introduced in chapter 6. The simple step
simulations of chapter 6 require order sorted signatures without multiply declared
operators, so simulation techniques which enable us to eliminate multiply declared operators will enhance the applicability of the techniques of chapter 6. In the next chapter two techniques will be presented for the elimination of multiply declared operators, one based on class step simulation and one on composed step simulation. Combining class step simulation or composed step simulation on the one hand and simple step simulation on the other hand forces us to say something about the result of this combination. The results are obvious and easy to understand. They are captured in the following two theorems.

**Theorem 3.19.** Composition of simple step simulation and class step simulation.

I. If \( \langle S'', R'' \rangle, \phi_2 \) is a simple step simulator of \( \langle S', R' \rangle \) and \( \langle S', R' \rangle, \phi_1 \) is a class step simulator of \( \langle S, R \rangle \) then \( \langle S'', R'' \rangle, \phi_1 \circ \phi_2 \) is a class step simulator of \( \langle S, R \rangle \).

II. If \( \langle S'', R'' \rangle, \phi_2 \) is a class step simulator of \( \langle S', R' \rangle \) and \( \langle S', R' \rangle, \phi_1 \) is a simple step simulator of \( \langle S, R \rangle \) then \( \langle S'', R'' \rangle, \phi_1 \circ \phi_2 \) is a class step simulator of \( \langle S, R \rangle \).

**Proof:** easy. \( \square \)

**Theorem 3.20.** Composition of simple step simulation and composed step simulation.

I. If \( \langle S'', R'' \rangle, \phi_2 \) is a simple step simulator of \( \langle S', R' \rangle \) and \( \langle S', R' \rangle, \phi_1, \psi \) is a composed step simulator of \( \langle S, R \rangle \) then \( \langle S'', R'' \rangle, \phi_1 \circ \phi_2, \phi_2 \circ \psi \) is a class step simulator of \( \langle S, R \rangle \).

II. If \( \langle S'', R'' \rangle, \phi_2, \psi \) is a composed step simulator of \( \langle S', R' \rangle \) and \( \langle S', R' \rangle, \phi_1 \) is a simple step simulator of \( \langle S, R \rangle \) then \( \langle S'', R'' \rangle, \phi_1 \circ \phi_2, \psi \circ \phi_1 \) is a composed step simulator of \( \langle S, R \rangle \).

**Proof:** easy. \( \square \)

The composition of a class step simulator and a composed step simulator will in general result in something which is neither a class step simulator nor a composed step simulator.
Chapter 4

Resolution of Overloadedness

4.1 Introduction

Multiply declared operators (also called overloaded operators) can be resolved with the simulations that were introduced in the previous chapter. It appears that multiply declared operators can be resolved with class step simulation if the rewrite system is left linear and with composed step simulation if the typing of the multiply declared operators satisfies the pre-regularity requirement. Section 4.2 describes resolution by class step simulation, section 4.3 describes resolution by composed step simulation, and section 4.4 presents examples of both cases.

4.2 Resolution by Class Step Simulation

Given an order sorted system $\langle \Sigma, RS \rangle$, a plain order sorted system and a domain mapping $\langle \langle \Sigma', RS \rangle, \phi \rangle$ will be derived that is a class step simulator of $\langle \Sigma, RS \rangle$. The rules of $RS$ have to be left linear.

Definition 4.1. A labeling from order sorted signature $\Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle$ to plain order sorted signature $\Sigma' = \langle S, Rdecs, Odecs', Vdecs \rangle$ is a pair $\langle L_{decl}, L_{term} \rangle$ consisting of two functions:

- $L_{decl} : Odecs \rightarrow Odecs'$ (bijective),
- $L_{term} : T_\Sigma \rightarrow P(T_{\Sigma'})$,

such that:

- $L_{decl}(\langle op, \langle s_1...s_n, s_r \rangle \rangle) = \langle op, \langle s_1...s_n, s_r \rangle \rangle$ if $op$ is not multiply declared,
- $L_{term}(v) = \{v\}$ for all variables $v \in V$,
- $L_{term}(k) = \{k\}$ for all constants $k \in K$,
- $L_{term}(op(t_1, ..., t_n)) = \{op'(t_1', ..., t_n') \mid$
\[ \exists s_1 \ldots s_n \in S^*, s_r \in S \ [L_{\text{decl}}((\text{op}, \langle s_1 \ldots s_n, s_r \rangle)) = (\text{op}', \langle s_1 \ldots s_n, s_r \rangle)] \wedge \]
\[ \forall 1 \leq i \leq n \ [t'_i \in L_{\text{term}}(t_i) \wedge (t'_i \not\in V \Rightarrow t'_i \in T_{\Sigma'}(s_i))]. \]

Note that being plain of \( \Sigma' \) and being bijective of \( L_{\text{decl}} \) are essential components of the definition. A surjective function \( L_{\text{oper}} : O_{\Sigma'} \to O_{\Sigma} \) can be derived from \( L_{\text{decl}} \) as follows:

\[ L_{\text{oper}}(\text{op}') = \text{op} \Leftrightarrow \exists s_1 \ldots s_n \in S^*, s_r \in S \]
\[ [L_{\text{decl}}((\text{op}, \langle s_1 \ldots s_n, s_r \rangle)) = (\text{op}', \langle s_1 \ldots s_n, s_r \rangle)]. \]

Obviously each \( L_{\text{decl}} \) uniquely determines an \( L_{\text{oper}} \) (under the given requirements) and vice versa.

Intuitively: a labeling is a mapping in which every multiply declared operator gets a number of distinct labels which refer to specific declarations. The association between specific labels and specific declarations is fixed in \( L_{\text{decl}} \). The function \( L_{\text{term}} \) assigns labels to multiply declared operators in terms in such a way that the only type violations in the resulting terms are because of variables (type violations of variables have a purpose that will become clear after example 4.5 and theorem 4.6).

**Theorem 4.2.** Let \( (L_{\text{decl}}, L_{\text{term}}) \) be a labeling from \( \Sigma \) to \( \Sigma' \). Then:

- \( L_{\text{decl}} \) is unique up to renaming of multiply declared operators,
- With a fixed \( L_{\text{decl}} \): \( L_{\text{term}} \) is uniquely determined.

**Proof:** easy.

**Definition 4.3.** The un-labeling function \( \phi : T_{\Sigma'} \to T_{\Sigma} \) for a labeling \( (L_{\text{decl}}, L_{\text{term}}) \) from \( \Sigma = (S, R\text{decs}, O\text{decs}, V\text{decs}) \) to \( \Sigma' = (S, R\text{decs}', O\text{decs}', V\text{decs}) \) is defined recursively as:

- \( \phi(v) = v \) for all \( v \in V \),
- \( \phi(k) = k \) for all \( k \in K \),
- \( \phi(\text{op}'(t'_1, \ldots, t'_n)) = (L_{\text{oper}}(\text{op}'))(\phi(t'_1), \ldots, \phi(t'_n)) \)

**Theorem 4.4.** Let \( (L_{\text{decl}}, L_{\text{term}}) \) be a labeling from \( \Sigma \) to \( \Sigma' \) and let \( \phi \) be the corresponding un-labeling function. Then:

1. \( \forall t' \in T_{\Sigma'} \ [\phi(t') = t \Leftrightarrow t' \in L_{\text{term}}(t)] \).
2. If \( t \in T_{\Sigma} \) then \( L_{\text{term}}(t) \cap T_{\Sigma'} \neq \emptyset \).
3. \( \phi \) is total and surjective,
4.2. RESOLUTION BY CLASS STEP SIMULATION

Proof. We will prove \( \Rightarrow \) with induction on the structure of terms.

Proof of \( \Rightarrow \). Suppose \( op'(t'_1, \ldots, t'_n) \) is an arbitrary term in \( T_{\Sigma'} \) and \( \phi(op'(t'_1, \ldots, t'_n)) = op(t_1, \ldots, t_n) \). From the definition of \( \phi \) and \( L_{\text{oper}} \) we know that there is a \( s_1 \ldots s_n \in S^n, s_r \in S \) such that \( L_{\text{decl}}(\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle) = \langle op', \langle s_1 \ldots s_n, s_r \rangle \rangle \) and \( \phi(t'_i) = t_i \) for all \( 1 \leq i \leq n \). From the definition of well-formed terms we know that \( t'_i \in T_{\Sigma'}(s_i) \) for all \( 1 \leq i \leq n \). The induction hypothesis now implies that \( t'_i \in L_{\text{term}}(t_i) \) for all \( 1 \leq i \leq n \). It is easy to see now that \( op'(t'_1, \ldots, t'_n) \in L_{\text{term}}(op(t_1, \ldots, t_n)) \): there is an appropriate \( s_1 \ldots s_n \) and an \( s_r \) and the condition \( \forall 1 \leq i \leq n [t'_i \in L_{\text{term}}(t_i) \land (t'_i \notin V \Rightarrow t'_i \in T_{\Sigma'}(s_i))] \) is easily seen to hold: \( t'_i \in T_{\Sigma'} \) regardless of whether or not \( t'_i \in V \).

Proof of \( \Leftarrow \). Suppose \( op'(t'_1, \ldots, t'_n) \) is an arbitrary term in \( T_{\Sigma'} \) and \( op'(t'_1, \ldots, t'_n) \in L_{\text{term}}(op(t_1, \ldots, t_n)) \). From the definition of \( L_{\text{term}} \) we know that there is \( s_1 \ldots s_n \in S^n, s_r \in S \) such that \( L_{\text{decl}}(\langle op(s_1 \ldots s_n, s_r) \rangle) = \langle op', \langle s_1 \ldots s_n, s_r \rangle \rangle \). Moreover, the definition of \( L_{\text{term}} \), together with the well-formedness of \( op'(t'_1, \ldots, t'_n) \) implies that \( \forall 1 \leq i \leq n [t'_i \in L_{\text{term}}(t_i) \land t'_i \in T_{\Sigma'}(s_i)] \). The induction hypothesis implies that \( \phi(t'_i) = t_i \) for all \( 1 \leq i \leq n \). It is easy to see now that \( \phi(op'(t'_1, \ldots, t'_n)) = op(t_1, \ldots, t_n) \).

The second item of this theorem is a direct consequence of the definition of well-formed terms and the third item is a consequence of the first and the second. \( \square \)

One of the tasks to be accomplished is to provide enough rules to rewrite each representative of a certain class of terms to a representative of the rewrite related class. This cannot be done by simply taking all possible well-formed labelings of the left hand sides of a rule to be transformed, to form left hand sides of rules for the new domain, because sometimes variables have a type which is too large. An example to illustrate this.

**Example 4.5.** Problems with labeling and types of variables. Consider an order sorted signature \( \Sigma = \langle S, R\text{decs}, O\text{decs}, V\text{decs} \rangle \) with:

- \( \{s_1, s_2\} \subseteq S; \)
- \( s_1 < s_2; \)
- \( \{\langle f, \langle s_1, s_1 \rangle \rangle, \langle f, \langle s_2, s_2 \rangle \rangle, \langle c, \langle e, s_1 \rangle \rangle \} \subseteq O\text{decs}; \)
- \( \langle v, s_2 \rangle \in V\text{decs}; \)

and consider a labeling \( \langle L_{\text{decl}}, L_{\text{term}} \rangle \) from \( \Sigma \) to \( \Sigma' \) with \( L_{\text{decl}} \) such that:

- \( L_{\text{decl}}(\langle f, \langle s_1, s_1 \rangle \rangle) = \langle f_1, \langle s_1, s_1 \rangle \rangle; \)
- \( L_{\text{decl}}(\langle f, \langle s_2, s_2 \rangle \rangle) = \langle f_2, \langle s_2, s_2 \rangle \rangle. \)

Obviously \( \{f(v), f(c)\} \subseteq T_{\Sigma} \) and \( f(v) \) matches in \( f(c) \). However, there is a well-formed labeling of \( f(c) \), notably \( f_1(c) \), in which no well-formed labeling of \( f(v) \) matches \( f_1(v) \) is not well-formed because the type of \( v \) is too large. \( \square \)
**Theorem 4.6.** Let $t, l \in T_{\Sigma}$ be two terms, $l$ linear. Let $\langle L_{\text{decl}}, L_{\text{term}} \rangle$ be a labeling from $\Sigma$ to plain $\Sigma'$ and let $\ell' \in L_{\text{term}}(l) \cap T_{\Sigma'}$. Now $l$ matches in $t$ at position $p \in Pos(t)$ if and only if there is an $\ell' \in L_{\text{term}}(l)$ and a variable substitution $\alpha$ such that $\ell'^{\alpha} \in T_{\Sigma'}$ and $\ell'^{\alpha}$ matches in $\ell'$ at position $p$.

**Proof of $\Rightarrow$.** It is easy to see that $\{ \ell'^{\alpha} | \ell' \in L_{\text{term}}(l) \cap T_{\Sigma'} \} \subseteq L_{\text{term}}(l|_p) \cap T_{\Sigma'}$, so it is sufficient to prove that the proposition holds for $p = \varepsilon$. We have to show that if there is a substitution $\sigma : V_{\Sigma} \rightarrow T_{\Sigma}$ such that $\ell'^{\sigma} = \ell$, then there is a substitution $\sigma' : V_{\Sigma'} \rightarrow T_{\Sigma'}$, an $\ell' \in L_{\text{term}}(l)$ and a variable substitution $\alpha : V_{\Sigma} \rightarrow V_{\Sigma'}$ such that $\ell'^{\alpha}$ is well-formed and $(\ell'^{\alpha})^{\sigma'} = \ell'$. Well, let:

- $VP = \{ \langle v, p \rangle \in V_{\Sigma} \times N^* | p \in Pos(l), l|_p = v \}$,
- $\ell' = t'[v_1], ..., [v_n], p_n$ with $\langle v_i, p_i \rangle$ ranging over $VP$,
- $\alpha = \{ v_i \leftarrow v'_i | \langle v_i, p_i \rangle \in VP, v'_i \text{ a new variable,} \}$
- $Type(v'_i) = Type(t'|_{p_i})$,
- $\sigma' = \{ v'_i \leftarrow t'|_{p_i} | \langle v_i, p_i \rangle \in VP, \alpha(v_i) = v'_i \}$.

The set $VP$ is the set of variable positions in $l$. The term $\ell'$ is well-defined because the positions $p_i$ are independent, i.e. none of the $p_i$ is a prefix of any $p_j$, this means that the order of applying subterm replacements is irrelevant for the final outcome. It is easy to see that $\ell' \in L_{\text{term}}(l)$: obviously $t'[v_1], ..., [v_n], p_n = l$ and $\phi(t'[v_1], ..., [v_n], p_n) = t[v_1], ..., [v_n], p_n$, now by theorem 4.4: $\ell' \in L_{\text{term}}(l)$. Clearly $\ell'^{\alpha}$ is a well-formed term: since $\ell'$ is well-formed, $Type(t'|_{p_i})$ is an allowed type at position $p \in Pos(t')$ (note that in a plain ordered signature as $\Sigma'$ each term has one unambiguously defined minimal type), therefore $\ell'^{\alpha}$ is well-formed. The fact that $\sigma'$ is a right (non-type-increasing) substitution and that $(\ell'^{\alpha})^{\sigma'} = \ell'$ is now evident. Note that the proof cannot be applied to non-linear patterns $l$ because $VP$ would then contain multiple entries for one single variable and $\alpha$ would not be defined well.

**Proof of $\Leftarrow$.** This direction of the equivalence is almost trivial. The implication holds because substitutions, and in particular variable substitutions, are always non-type-increasing. It is easy to see therefore that a pattern that matches in a labeled term matches also in the unlabeled term.

**Algorithm 4.7.** A procedural description will be given of how to derive a rule set for a class step simulating system $\langle (\Sigma', RS'), \phi \rangle$ from a left linear system $\langle \Sigma, RS \rangle$. It is assumed that $\langle L_{\text{decl}}, L_{\text{term}} \rangle$ is a labeling from $\Sigma$ to $\Sigma'$. The algorithm will be stated in imperative programming style; the variable `workarea` can be bound to sets of rules. The $\lfloor \cdot \rfloor_\alpha$-operator was introduced in definition 2.20.

Set `workarea` to $\emptyset$;
For each rule $h = \langle l, r \rangle \in RS$ Do:
4.3. RESOLUTION BY COMPOSED STEP SIMULATION

Let $r'$ be some particular element of $L_{\text{term}}(r)$;
Let $L' = \{(l', \alpha) \in L_{\text{term}}(l) \times (V_{\Sigma'} \rightarrow V_{\Sigma'}) \mid$
\[l'^{\alpha} \in T_{\Sigma'},\]
\[|\text{Var}(l'^{\alpha})| = |\text{Var}(l')|,\]
all variables in $l'^{\alpha}$ are maximally typed\}$_{\alpha}$;

Add to \textit{workarea} the set $\{(l'^{\alpha}, r'^{\alpha}) \mid (l', \alpha) \in L'\}$.

End For-iteration over rules;
Set $RS'$ to \textit{workarea}. □

Theorem 4.8. Let $(\Sigma, RS)$ be a left linear order sorted system, $(L_{\text{decl}}, L_{\text{term}})$ a labeling from $\Sigma$ to $\Sigma'$ with $\phi$ the corresponding un-labeling function. Let $RS'$ be the rule-set which is derived from $(\Sigma, RS)$ and $(L_{\text{decl}}, L_{\text{term}})$ by algorithm 4.7. Now $(\langle \Sigma', RS' \rangle, \phi)$ is a class step simulator of $(\Sigma, RS)$.

\textit{Proof:} first the observation that the $\text{\_}_{\alpha}$-operator in the algorithm for $RS'$ is for reasons of efficiency. Two rules which are alpha convertible into each other have both separately and both together exactly the same effect on terms to be rewritten, so $\text{\_}_{\alpha}$ can be ignored in the proof. For a similar reason the requirement of $l'^{\alpha}$ having all its variables maximally typed and the number of different variables not decreased, can be neglected. The operation of a rule $\langle l'^{\alpha'}, r'^{\alpha'} \rangle$ in which not all variables are maximally typed, or in which some variables have been identified, is entirely covered by a rule $\langle l'^{\alpha}, r'^{\alpha} \rangle$ in which all variables are maximally typed and in which the number of different variables is same as in $l'$. The domain mapping $\phi$ was already established to be total and surjective (theorem 4.4).

\textit{Proof of}$\Rightarrow$. Let $t_1, t_2 \in T_{\Sigma}$ and $t_1 \rightarrow_{RS} t_2$. There is a rule $\langle l, r \rangle \in RS$ and a position $p \in \text{Pos}(t_1)$ such that $l$ matches in $t_1$ at $p$. Clearly $\Phi^{-1}(t_1) = L_{\text{term}}(t_1) \cap T_{\Sigma'}$, so according to theorem 4.6 there is an $l' \in L_{\text{term}}(l)$ and a variable substitution $\alpha$ such that $l'^{\alpha}$ is well-formed and matches in $t'_1$ at $p$, where $t'_1 \in \Phi^{-1}(t_1)$. So $t'_1$ rewrites to $t'_2 = t'_1[(r'^{\alpha})^p]$, which can easily be seen to be a member of $\Phi^{-1}(t_2)$.

\textit{Proof of}$\Leftarrow$. Let $t'_1, t'_2 \in T_{\Sigma'}$ and $t'_1 \rightarrow_{RS'} t'_2$. There is a rule $\langle l', r' \rangle \in RS'$ and a position $p \in \text{Pos}(t'_1)$ such that $l$ matches in $t'_1$ at $p$. Obviously $\phi(l')$ matches in $\phi(t'_1)$ at $p$ and $\phi(t'_1)$ rewrites to $\phi(t'_2)$ based on the rule $\langle \phi(l'), \phi(r') \rangle$. From the construction of $RS'$ and theorem 4.6 follows that $RS$ contains an $\langle l, r \rangle$ with $l$ and $r$ such that $l'^{\alpha} = \phi(l')$ and $r'^{\alpha} = \phi(r')$ for some variable substitution $\alpha$. Therefore $t_1 \rightarrow_{RS} t_2$. □

4.3 Resolution by Composed Step Simulation

Given an order sorted system $\Sigma, RS$, a plain order sorted system an a domain mapping $(\langle \Sigma', RS \rangle, \phi)$ will be derived, that is a \textit{composed step simulator} of $(\Sigma, RS)$. The signature $\Sigma$ has to fulfill the pre-regularity requirement (definition 2.11). Pre-regularity is sufficient for our purpose. In a pre-regular signature each term has one unambiguously defined least type [21].
Definition 4.9. The least labeling function \( \psi : \Sigma \rightarrow \Sigma \) for a labeling \( \langle L_{\text{deq}}, L_{\text{term}} \rangle \) from pre-regular \( \Sigma = \langle S, R\text{decs}, O\text{decs}, V\text{decs} \rangle \) to plain \( \Sigma' = \langle S, R\text{decs}, O\text{decs}', V\text{decs} \rangle \) is defined recursively as:

- \( \psi(v) = v \) for all \( v \in V \),
- \( \psi(k) = k \) for all \( k \in K \),
- \( \psi(op(t_1, ..., t_n)) = op'(\psi(t_1), ..., \psi(t_n)) \)

with \( s_1...s_n \in S^n \) and \( s_r \in S \) such that \( t_1 \in \Sigma(s_1), ..., t_n \in \Sigma(s_n), L_{\text{deq}}(op, \langle s_1...s_n, s_r \rangle) = \langle op', \langle s_1...s_n, s_r \rangle \rangle \) and there is no \( op, \langle s_1'...s_n', s_r' \rangle \in O\text{decs} \) with \( t_1 \in \Sigma(s_1'), ..., t_n \in \Sigma(s_n') \) and \( s_r, s_1...s_n <_{\text{lex}} s_r, s_1...s_n \) (\( <_{\text{lex}} \) the lexicographic extension of \( < \), with the left most type the most significant type, note that this ordering assumes at least comparability of all corresponding types).

Obviously \( \psi(t) \in L_{\text{term}}(t) \cap \Sigma \) for all \( t \in \Sigma \).

\[ \square \]

Theorem 4.10. Let \( t, l \in T \) be two terms. Let \( \langle L_{\text{deq}}, L_{\text{term}} \rangle \) be a labeling from pre-regular \( \Sigma = \langle S, R\text{decs}, O\text{decs}, V\text{decs} \rangle \) to plain \( \Sigma = \langle S, R\text{decs}, O\text{decs}', V\text{decs} \rangle \) and let \( \psi \) be the corresponding least labeling function. Now \( l \) matches in \( t \) at position \( p \in \text{Pos}(t) \) if and only if there is a variable substitution \( \alpha : V_{\Sigma} \rightarrow V_{\Sigma} \) such that \( \psi(l^\alpha) \) matches in \( \psi(t) \) at \( p \).

Proof of \( \Rightarrow \). it is easy to see that \( \psi(t)|_p = \psi(t)|_p \), so it is sufficient to prove that the proposition holds for \( p = \epsilon \). We have to show that if there is a substitution \( \sigma : V_{\Sigma} \rightarrow \Sigma \) such that \( l^\sigma = t \) then there is a substitution \( \sigma' : V_{\Sigma'} \rightarrow \Sigma \) and a variable substitution \( \alpha : V_{\Sigma} \rightarrow V_{\Sigma} \) such that \( (\psi(l^\alpha))^{\sigma' = \psi(t)} \). Well, let:

- \( V_P = \{ (v, p) \in V \times N \mid p \in \text{Pos}(l), l|_p = v \} \),
- \( \alpha = \{ v_i \leftarrow v_i' \mid v_i \in V \} \),
  - each \( v_i' \) is a new variable (for each distinguished \( v_i \)),
  - \( \exists p_i \in N^*[\langle v_i, p_i \rangle \in VP] \),
  - \( \forall 1 \leq j \leq |VP| \mid \text{Type}(v_j') = \text{Type}(\psi(t)|_{p_j}) \}, \)
- \( \sigma' = \{ v_i' \leftarrow \psi(t)|_{p_i} \mid (v_i, p_i) \in VP, \alpha(v_i) = v_i' \} \).

The set \( VP \) is the set of variable positions in \( l \). Note that because of non-linearity of \( l \) there may be more than one entry for certain variables. The well-definedness of \( \alpha \) follows from the fact that if there are multiple positions for a certain variable, the sub-terms in \( \psi(t) \) at these positions are equal (because \( l \) matches in \( t \) at \( \epsilon \)) and therefore the type of their sub-terms is also equal; \( v_i' \in \Sigma \text{Type}(v_i) \) because \( \psi(t)|_p \in \Sigma \text{Type}(v_i) \). The fact that \( \sigma' \) is well-defined and that \( (\psi(l^\alpha))^{\sigma' = \psi(t)} \) is now evident.
4.3. RESOLUTION BY COMPOSED STEP SIMULATION

Proof of \( \Leftarrow \). This direction of the equivalence is almost trivial. The implication holds because substitutions, and in particular variable substitutions, are always non-type-increasing. It is easy to see therefore that a pattern that matches in a labeled term matches also in the unlabeled term. \( \square \)

Algorithm 4.11. A procedural description will be given of how to derive a rule set for a composed step simulating system \( \langle \Sigma', RS' \rangle, \phi \rangle \) from a left linear system \( \langle \Sigma, RS \rangle \). It is assumed that \( L_{\text{decl}}, L_{\text{term}} \rangle \) is a labeling from \( \Sigma \) to \( \Sigma' \) and \( \psi \) is a least labeling function for this labeling. The algorithm will be stated in imperative programming style; the variables main-area and aux-area can be bound to sets of rules. The \( [\cdot]_\alpha \)-operator was introduced in definition 2.20.

For all rules \( h = \langle l, r \rangle \in RS \) Do:

Let \( L' = \{ \langle (l, \alpha) \mid \alpha : V_\Sigma \rightarrow V_{\Sigma'}, |\text{Var}(l^\alpha)| = |\text{Var}(l)|, \alpha \text{ variables in } \psi(l^\alpha) \text{ are maximally typed} \} \}_{\alpha} \);

Add to main-area the set \( \{ \langle \psi(l^\alpha), \psi(r^\alpha) \rangle \mid \langle l, \alpha \rangle \in L' \} \).

End of For-iteration over rules;

For all multiply declared operators \( op \in O_\Sigma \) Do:

For all pairs of declarations \( \langle op_1, \langle s_1, \ldots, s_n, s_r \rangle \rangle, \langle op_2, \langle t_1, t_n, t_r \rangle \rangle \) such that:

\[
L_{\text{decl}}(\langle op, \langle s_1, \ldots, s_n, s_r \rangle \rangle) = \langle op_1, \langle s_1, \ldots, s_n, s_r \rangle \rangle,
\]

\[
L_{\text{decl}}(\langle op, \langle t_1, \ldots, t_n, t_r \rangle \rangle) = \langle op_2, \langle t_1, \ldots, t_n, t_r \rangle \rangle,
\]

\( t_r, t_1, \ldots, t_n <_{\text{lex}} s_r, s_1, \ldots, s_n \)

Do:

Add to aux-area the rule: \( \langle op_1(v_1, \ldots, v_n), op_2(v_1, \ldots, v_n) \rangle \) with all variables \( v_1, \ldots, v_n \) distinct and maximally typed.

End of For-iteration over pairs of declarations.

End of For-iteration over multiply declared operators;

Set \( RS' \) to main-area \( \cup \) aux-area. \( \square \)

Theorem 4.12. Let \( \Sigma \) be a pre-regular signature, \( L_{\text{decl}}, L_{\text{term}} \rangle \) a labeling from \( \Sigma \) to plain \( \Sigma' \), \( \phi \) the corresponding un-labeling function, and \( \psi \) a least labeling function for this labeling. Let \( RS \) be a rewrite system with an irreflexive one step rewrite relation and \( RS' \) the rule-set which is derived from \( \langle \Sigma, RS \rangle \), \( \langle L_{\text{decl}}, L_{\text{term}} \rangle \), and \( \psi \) by algorithm 4.11. Now the triple \( \langle \Sigma', RS', \phi, \psi \rangle \) is a composed step simulator of \( \langle \Sigma, RS \rangle \).

Proof: Like in theorem 4.8 also here the observation that the \( [\cdot]_\alpha \)-operator and the requirement for a maximal number of different variables and maximal typedness of variables in \( l^\alpha \) is only for reasons of efficiency. These reductions and restrictions can be neglected in the proof. Let \( RS'_\text{main} \) be the set of rules which is the final contents of main-area and \( RS'_\text{aux} \) the set of rules which is the final contents of aux-area. Because of the irreflexivity of \( \rightarrow_{RS} \), \( RS'_\text{main} \) performs only rewrite steps between different classes. Obviously \( RS'_\text{aux} \) performs only rewrite steps within one class.
RS'\textsubscript{aux} terminates: assign a natural number weight to each operator in such a way that weight(op\textsubscript{2}) < weight(op\textsubscript{1}) if \( L_{\text{decl}}(\langle \text{op}, \langle s\textsubscript{1}...s\textsubscript{n}, s\textsubscript{r} \rangle \rangle) = \langle \text{op\textsubscript{1}}, \langle s\textsubscript{1}...s\textsubscript{n}, s\textsubscript{r} \rangle \rangle \), \( L_{\text{decl}}(\langle \text{op}, \langle t\textsubscript{1}...t\textsubscript{n}, t\textsubscript{r} \rangle \rangle) = \langle \text{op\textsubscript{2}}, \langle t\textsubscript{1}...t\textsubscript{n}, t\textsubscript{r} \rangle \rangle \) and \( t\textsubscript{r}, t\textsubscript{1}...t\textsubscript{n} \triangleleft_{\text{lex}} s\textsubscript{r}, s\textsubscript{1}...s\textsubscript{n} \). This is possible because \( \triangleleft_{\text{lex}} \) is a partial order on \( S^{n+1} \). Assign a zero weight to variables and define the weight of a term to be the sum of the weights of its variables and operators. The weight of each term is a natural number which is decreased by each rule in \( RS'\textsubscript{aux} \). Hence \( RS'\textsubscript{aux} \) terminates. The conditions of definition 3.11 can now be verified easily:

- Totality and injectivity of \( \psi \), as well as the requirement \( \phi(\psi(t)) = t \), can be proven straightforwardly with induction on term structure.

- \( \forall t\textsubscript{1} \in T_{\Sigma} \ [\forall t'\textsubscript{1} \in \Phi^{-1}(t\textsubscript{1}) \ [t'\textsubscript{1} \rightarrow_{RS'\textsubscript{aux}} \psi(t\textsubscript{1})]] \), because \( RS'\textsubscript{aux} \) terminates and for each \( t'\textsubscript{1} \) which does not equal \( \psi(t\textsubscript{1}) \) there is a rule in \( RS'\textsubscript{aux} \) that can be applied to \( t'\textsubscript{1} \).

- \( t\textsubscript{1} \rightarrow_{RS} t\textsubscript{2} \Rightarrow \exists t'\textsubscript{2} \in \Phi^{-1}(t\textsubscript{2}) \ [t'\textsubscript{1} \rightarrow_{RS'\textsubscript{main}} t'\textsubscript{2}] \) because if \( t\textsubscript{1} \) was rewritten, based on rule \( \langle l, r \rangle \in RS \) then \( \psi(l\alpha) \) matches in \( \psi(t\textsubscript{1}) \) for some \( \alpha : V_{\Sigma} \rightarrow V_{\Sigma} \) (theorem 4.10) and \( \psi(t\textsubscript{1}) \) is rewritten to some \( t'\textsubscript{2} \in \Phi^{-1}(t\textsubscript{2}) \) based on rule \( \langle \psi(l\alpha), \psi(r\alpha) \rangle \in RS'\textsubscript{main} \).

- \( \exists t'\textsubscript{1} \in \Phi^{-1}(t\textsubscript{1}) \ [t'\textsubscript{1} \rightarrow_{RS'\textsubscript{main}} t'\textsubscript{2}] \Rightarrow t\textsubscript{1} \rightarrow_{RS} t\textsubscript{2} \) because rewrite steps between classes are performed only by rules from \( RS'\textsubscript{main} \) and from the derivation process of \( RS'\textsubscript{main} \) and theorem 4.10 can be deduced that if \( t'\textsubscript{1} \rightarrow_{RS} t'\textsubscript{2} \) based on \( \langle \psi(l\alpha), \psi(r\alpha) \rangle \in RS'\textsubscript{main} \) then \( t\textsubscript{1} \rightarrow_{RS} t\textsubscript{2} \) based on \( \langle l, r \rangle \in RS \). \( \square \)

### 4.4 A Large Example

A large example will be presented in which most aspects of the previously described process of resolution of multiply declared operators are illustrated. An easy to understand syntax for specifying signatures and systems will be used.

The example is a definition of the operations addition, multiplication, and division by 2 in a system in which the natural numbers have two subtypes: the odd naturals and the even naturals. Specification OddEven1 is an order sorted system with multiple declarations. Resolution by class step simulation leads to plain order sorted specification OddEven2 and resolution by composed step simulation to plain order sorted specification OddEven3.

**Specification OddEven1**

- **Sorts**: odd, even, nat;
- **Relations**: odd < nat, even < nat;
- **Operations**: (5)
  - zero: even;
  - succ: nat → nat,
  - even → odd,
odd → even;
add: nat × nat → nat,
even × even → even,
even × odd → odd,
odd × even → odd,
odd × odd → even;
mult: nat × nat → nat,
even × nat → even,
nat × even → even,
odd × odd → odd;
div2: even → nat;

Vars: x₁, x₂: nat; y: even; (3)

Rules: (6)
add(zero, x₁) → x₁;
add(succ(x₁), x₂) → succ(add(x₁, x₂));
mult(zero, x₁) → zero;
mult(succ(x₁), x₂) → add(mult(x₁, x₂), x₂);
div2(zero) → zero;
div2(succ(succ(y))) → succ(div2(y));

End OddEven1

Specification OddEven1 has a rewrite system in which each rule is left linear and in which the declarations fulfill the pre-regularity requirement. Resolution of overloadedness is therefore possible with both alternatives: class step simulation and composed step simulation. The derivation of OddEven2 from OddEven1 is straightforward. Label all left hand side patterns in any possible way such that the variables in the patterns are maximally typed (eventually after one or more variables are replaced by “lesser” typed variables). Each distinct result should be a left hand side of a rule in the simulating system. The choice of the right hand sides of these rules offers some freedom. In this case we have chosen minimal typed right hand sides of rules. There are no rules for add₄ and add₅ with first operand zero because zero is known to be even. Same arguments for mult₄.

Specification OddEven2
Sorts: odd, even, nat;
Relations: odd < nat, even < nat;

Operations: (14)
zero: even;
succ₁: nat → nat;
succ₂: even → odd;
succ₃: odd → even;
add₁: nat × nat → nat;
add₂: even × even → even;
add₃: even × odd → odd;
CHAPTER 4. RESOLUTION OF OVERLOADEDNESS

add₄: odd × even → odd;
add₅: odd × odd → even;
mult₁: nat × nat → nat;
mult₂: even × nat → even;
mult₃: nat × even → even;
mult₄: odd × odd → odd;
div₂: even → nat;

Vars: x₁, x₂: nat; y₁, y₂: even; z₁, z₂: odd; (6)

Rules: (23)

add₁(zero, x₁) → x₁;
add₂(zero, y₁) → y₁;
add₃(zero, z₁) → z₁;
add₁(succ₁(x₁), x₂) → succ₁(add₁(x₁, x₂));
add₁(succ₂(y₁), x₁) → succ₁(add₁(y₁, x₁));
add₁(succ₃(z₁), x₁) → succ₁(add₁(z₁, x₁));
add₂(succ₂₂(z₁), y₁) → succ₁(add₂(z₁, y₁));
add₃(succ₂₂(z₁), z₂) → succ₂(add₂(z₁, z₂));
add₄(succ₂₂(y₁), y₂) → succ₂2(add₂(y₁, y₂));
add₅(succ₂₂(y₁), z₁) → succ₂(add₂(y₁, z₁));
mult₁(zero, x₁) → zero;
mult₄(zero, x₁) → zero;
mult₃(zero, y₁) → zero;
mult₁(succ₁(x₁), x₂) → add₁(mult₁(x₁, x₂), x₂);
mult₁(succ₂(y₁), x₁) → add₁(mult₂(y₁, x₁), x₁);
mult₁(succ₃(z₁), y₁) → add₁(mult₁(z₁, y₁), x₁);
mult₂(succ₁(x₁), y₁) → add₁(mult₂(z₁, y₁), y₁);
mult₄(succ₁(y₁), y₂) → add₂(mult₂(y₁, y₂), y₂);
mult₂(succ₃(z₁), y₁) → add₂(mult₂(z₁, y₁), y₁);
mult₃(succ₂₂(y₁), z₁) → add₃(mult₂(z₁, y₁), z₁);
div₂(succ₂₂(y₁)) → succ₁(div₂(y₁));
div₂(zero) → zero;

End OddEven2

The derivation of OddEven3 is also straightforward. According to the transformation descriptions OddEven3 has to contain same rules as OddEven2, except that rules in which operand and result types of two adjacent operators are not equal have been removed and rules for auxiliary rewrite steps have been added (two adjacent operators being two operators which have a parent-child relationship). Rules in which operand and result types of adjacent operators are not equal are:

add₁(zero, x₁) → x₁;
add₁(succ₂(y₁), x₁) → succ₁(add₁(y₁, x₁));
add₁(succ₃(z₁), x₁) → succ₁(add₁(z₁, x₁));
4.4. A LARGE EXAMPLE

\[
\begin{align*}
\text{mult}_1(\text{zero}, x_1) & \rightarrow \text{zero}; \\
\text{mult}_3(\text{zero}, y_1) & \rightarrow \text{zero}; \\
\text{mult}_2(\text{succ}_2(y_1), y_2) & \rightarrow \text{add}_2(\text{mult}_2(y_1, y_2), y_2); \\
\text{mult}_3(\text{succ}_3(z_1), y_1) & \rightarrow \text{add}_2(\text{mult}_3(z_1, y_1), y_1);
\end{align*}
\]

as can be verified easily (all these rules have \texttt{odd}- or \texttt{even}- typed operators at \texttt{nat}-typed operand positions). The auxiliary rules that have to be added are:

\[
\begin{align*}
\text{succ}_1(y_1) & \rightarrow \text{succ}_2(y_1); \\
\text{succ}_1(z_1) & \rightarrow \text{succ}_3(z_1); \\
\text{add}_1(y_1, y_2) & \rightarrow \text{add}_2(y_1, y_2); \\
\text{add}_1(y_1, z_1) & \rightarrow \text{add}_3(y_1, z_1); \\
\text{add}_1(z_1, y_1) & \rightarrow \text{add}_4(z_1, y_1); \\
\text{add}_1(z_1, z_2) & \rightarrow \text{add}_5(z_1, z_2); \\
\text{mult}_1(y_1, x_1) & \rightarrow \text{mult}_2(y_1, x_1); \\
\text{mult}_1(x_1, y_1) & \rightarrow \text{mult}_3(x_1, y_1); \\
\text{mult}_1(y_1, y_2) & \rightarrow \text{mult}_2(y_1, y_2); \\
\text{mult}_3(y_1, y_2) & \rightarrow \text{mult}_4(y_1, y_2); \\
\text{mult}_3(z_1, z_2) & \rightarrow \text{mult}_4(z_1, z_2);
\end{align*}
\]

Specification \texttt{OddEven3} becomes:

\textbf{Specification \texttt{OddEven3}}

\textbf{Sorts, Relations, Operations, Vars: like in \texttt{OddEven2}}

\textbf{Rules: (27)}

\[
\begin{align*}
\text{add}_2(\text{zero}, y_1) & \rightarrow y_1; \\
\text{add}_3(\text{zero}, z_1) & \rightarrow z_1; \\
\text{add}_4(\text{succ}_1(x_1), x_2) & \rightarrow \text{succ}_1(\text{add}_1(x_1, x_2)); \\
\text{add}_2(\text{succ}_3(z_1), y_1) & \rightarrow \text{succ}_3(\text{add}_4(z_1, y_1)); \\
\text{add}_3(\text{succ}_3(z_1), z_2) & \rightarrow \text{add}_3(\text{add}_4(z_1, z_2)); \\
\text{add}_4(\text{add}_2(y_1, y_2) & \rightarrow \text{add}_2(\text{add}_2(y_1, y_2)); \\
\text{add}_5(\text{succ}_2(y_1), z_1) & \rightarrow \text{succ}_3(\text{add}_4(y_1, z_1)); \\
\text{mult}_2(\text{zero}, x_1) & \rightarrow \text{zero}; \\
\text{mult}_1(\text{succ}_1(x_1), x_2) & \rightarrow \text{add}_1(\text{mult}_1(x_1, x_2), x_2); \\
\text{mult}_1(\text{succ}_2(y_1), x_1) & \rightarrow \text{add}_1(\text{mult}_2(y_1, x_1), x_1); \\
\text{mult}_1(\text{succ}_3(z_1), x_1) & \rightarrow \text{add}_1(\text{mult}_3(z_1, x_1), x_1); \\
\text{mult}_2(\text{succ}_3(z_1), x_1) & \rightarrow \text{add}_1(\text{mult}_4(z_1, x_1), x_1); \\
\text{mult}_3(\text{succ}_1(x_1), y_1) & \rightarrow \text{add}_2(\text{mult}_3(x_1, y_1), y_1); \\
\text{mult}_3(\text{succ}_2(y_1), z_1) & \rightarrow \text{add}_3(\text{mult}_2(y_1, z_1), z_1); \\
\text{div}_2(\text{succ}_3(\text{succ}_2(y_1)) & \rightarrow \text{add}_2(\text{div}_2(y_1)); \\
\text{div}_2(\text{zero}) & \rightarrow \text{zero}; \\
\text{succ}_1(y_1) & \rightarrow \text{succ}_2(y_1); \\
\text{succ}_1(z_1) & \rightarrow \text{succ}_3(z_1); \\
\text{add}_1(y_1, y_2) & \rightarrow \text{add}_2(y_1, y_2); \\
\text{add}_1(y_1, z_1) & \rightarrow \text{add}_3(y_1, z_1); \\
\end{align*}
\]
add₁(z₁, y₁) → add₄(z₁, y₁);
add₁(z₁, z₂) → add₅(z₁, z₂);
mult₁(y₁, x₁) → mult₂(y₁, x₁);
mult₃(x₁, y₁) → mult₃(x₁, y₁);
mult₁(y₁, y₂) → mult₂(y₁, y₂);
mult₃(y₁, y₂) → mult₄(y₁, y₂);
mult₁(z₁, z₂) → mult₄(z₁, z₂);

End OddEven3
Chapter 5

Retrenched Vocabularies

5.1 Introduction

From this chapter and onwards, attention will be focused on a new basic formalism for the construction of terms and on one particular kind of step simulation: simple step simulation (definition 3.5). The merit of the simulations of chapter 4 is a wider applicability of the theory to be introduced in the chapters to come. This theory is stated in terms of a new basic formalism for terms, the *retrenched vocabulary formalism*, which is roughly spoken equivalent with the order sorted vocabulary formalism *without* multiply declared operators. The resolution schemes of chapter 4 will extend the scope of the theory to general order sorted rewrite systems (*with* multiply declared operators).

The chapters 5 through 9 all deal with some particular simple step simulations and corresponding transformations. The order sorted signature formalism is not ideally suited to underly the description of these transformations because of lacking functionality and redundancy:

1. the absence of a primitive to restrict the type of the root operator of terms,
2. the more or less superfluous presence of explicit types in the signature.

To overcome these deficiencies a new basic formalism for the construction of terms will be introduced: the *retrenched vocabulary formalism*. The verb "to retrench" means: to cut of branches. The name is based on the close correspondence between terms and (mathematical) trees. This new basic formalism is in fact an intermediate form between the order sorted signature formalism and the context free grammar formalism. It is relatively easy to convert order sorted signatures into retrenched vocabularies and vice versa and also to convert retrenched vocabularies into context free grammars and vice versa.

The retrenched vocabulary formalism will be defined formally in section 5.2. In section 5.3 the definitions for term analysis of section 2.3 are adjusted for the new basic formalism. The conversion between vocabularies and signatures is subject of section 5.4 and the conversion between vocabularies and grammars is subject of
section 5.5. A justification for the introduction of a new basic formalism for the
construction of terms will be given in chapter 11. Retrenched vocabularies will
mostly be abbreviated to just “vocabularies” in the rest of this thesis.

5.2 The Formal Definition

**Definition 5.1.** A retrenched vocabulary is a 6-tuple \( (C, V, S, \alpha, \rho, \mu) \), with:

\[
\begin{align*}
C & : \text{the set of constructors}, \\
V & : \text{the set of variables}, \\
S & : \text{the root restriction set}, \\
\alpha & : C \to \mathbb{N}, \text{the arity function}, \\
\rho & : C \times \mathbb{N}^+ \to \mathcal{P}(C), \text{the restriction function}, \\
\mu & : V \to \mathcal{P}(C), \text{the match function},
\end{align*}
\]

for which the following six conditions hold. The sets \( YC, YV, \) and \( Y \), which are used in the conditions, can be derived from one or more components of the vocabulary and will be defined afterwards.

1. \( \forall c \in C, \forall 1 \leq n \leq \alpha(c) \ [\rho(c, n) \subseteq C] \): a constructor restriction is defined to be a subset of the set of constructors,

2. \( \forall c \in C, \forall n > \alpha(c) \ [\rho(c, n) = \perp] \): the restriction function is not defined for values of its second argument greater than the arity in question,

3. \( \forall C_s \in YV \ [\{v \in V \mid \mu(v) = C_s\} \text{ is a countably infinite set }] \), with \( YV \) as defined below: each subset \( C_s \) of \( C \) that acts as a representative of a type in the variable set has countably infinite associated variables,

4. \( YV \supseteq YC \land YV \ni S \), with \( YV \) and \( YC \) as defined below: for each represented type that acts as function value of \( \rho \), and for the “root type”, there is an associated variable,

5. \( \forall yv \in YV \ [\exists yc \in YC [yv \subseteq yc] \lor yv \subseteq S] \), with \( YV \) and \( YC \) as defined below: all variables fit somewhere in well-formed terms (definition 5.7),

6. \( S \subseteq C \): the root restriction set is defined to be a subset of the set of constructors.

A number of useful sets can be derived from the 6-tuple. They are defined as follows:

- \( K = \{c \in C \mid \alpha(c) = 0\} \): the subset of zero arity constructors, also called constants,
5.2. THE FORMAL DEFINITION

- $YC = \{ \rho(c,i) \mid c \in C \land 1 \leq i \leq \alpha(c) \}$: the set of represented types in the constructor set,

- $YV = \{ \mu(v) \mid v \in V \}$: the set of represented types in the variable set,

- $Y = YC \cup YV \cup \{ S \}$: the set of represented types (which is equal to $YV \cup \{ S \}$ due to condition 4).

The derived sets and the primary elements of the tuple will be indexed with a vocabulary specifier whenever ambiguity can arise, so for instance $Y_A$ and $\alpha_A$ pertain to vocabulary $A = (C, V, S, \alpha, \rho, \mu)$. A vocabulary will be called finite if and only if its constructor set $C$ is finite. In the rest of this thesis vocabularies are assumed to be finite.

The basic building blocks of terms in the sorted signature formalism, operators, will be called constructors in the vocabulary formalism. The reason is that the distance with the operator and the operation notion has become larger after the elimination of explicit types. There is a well-defined meaning for the constructor notion and a well-understood difference between operators and constructors in certain scientific works [16, 47, 55]. We do not make that difference. We just use the word constructor because we "construct" terms (all terms) with them.

In the definitions 5.2 through 5.8 the term concept based on the new formalism will be defined formally. The set of ground terms $TG$, the set of restricted terms $R$, the set of restricted ground terms $RG$, the set of well-formed terms $T$, and the set of well-formed ground terms $TG$ over a certain vocabulary will be defined as subsets of the set of terms $T$. For the definition of $R$ and $T$ an auxiliary operator $\text{Root}$ that extracts the root constructor of a term is needed. The set $R$ is in fact an auxiliary set for the definition of the set $T$. Ground terms are terms without variables and well-formed terms are terms that satisfy the restriction function and the root restriction set. The sets $TG$ and $R$, but also the sets $TG$ and $T$ may have a non-empty intersection. These intersections will be named well formed restricted terms and well-formed ground terms.

**Definition 5.2.** The set $T$ of terms over vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ is defined inductively as the smallest set such that:

- $K \subset T$;
- $V \subset T$;
- $\forall f \in C - K, t_1 \in T, \ldots, t_{\alpha(f)} \in T \ [f(t_1, \ldots, t_{\alpha(f)}) \in T]$.

**Definition 5.3.** The set $TG$ of ground terms over vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ is defined inductively as the smallest set such that:

- $K \subset TG$;
\( \forall f \in C - K, t_1 \in TG, \ldots, t_{\alpha(f)} \in TG \ [f(t_1, \ldots, t_{\alpha(f)}) \in TG]. \)

**Definition 5.4.** The Root operator extracts the root constructor from a given term: Root : \( T \to C \cup V \). Definition by case:

- Root\((k) = k\), for all \( k \in K \);
- Root\((v) = v\), for all \( v \in V \);
- Root\((f(t_1, \ldots, t_{\alpha(f)})) = f\), for all \( f \in C - K, t_1, \ldots, t_{\alpha(f)} \in T \).

**Definition 5.5.** The set \( \mathcal{R} \) of restricted terms over vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) is defined inductively as the smallest set such that:

- \( K \subset \mathcal{R} \);
- \( V \subset \mathcal{R} \);
- \( \forall f \in C - K, t_1 \in \mathcal{R}, \ldots, t_{\alpha(f)} \in \mathcal{R} \ [\forall 1 \leq i \leq \alpha(f)] \)
  
  \( [\text{if } t_i \in V \text{ then } \mu(t_i) \subseteq \rho(f, i) \text{ else } \text{Root}(t_i) \in \rho(f, i)] \Rightarrow \)

  \( f(t_1, \ldots, t_{\alpha(f)}) \in \mathcal{R} ] \).

**Definition 5.6.** The set \( \mathcal{RG} \) of restricted ground terms over vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) is defined as:

\( \mathcal{RG} = \mathcal{R} \cap TG. \)

**Definition 5.7.** The set \( \mathcal{T} \) of well-formed terms over vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) is defined as:

\( \mathcal{T} = \{ t \in \mathcal{R} \mid \text{if } t \in V \text{ then } \mu(t) \subseteq S \text{ else } \text{Root}(t) \in S \}. \)

**Definition 5.8.** The set \( \mathcal{TG} \) of well-formed ground terms over vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) is defined as:

\( \mathcal{TG} = \mathcal{T} \cap TG. \)

The notations for term sets \( T, TG, \mathcal{R}, \mathcal{RG}, \mathcal{T}, \mathcal{TG} \), will be indexed with vocabulary specifiers whenever ambiguity can arise, so \( T_A, TG_A, \mathcal{R}_A, \mathcal{RG}_A, \mathcal{T}_A, \mathcal{TG}_A \), are sets of terms over vocabulary \( V \). Because \( T \supseteq TG, T \supseteq \mathcal{R}, T \supseteq \mathcal{RG}, T \supseteq \mathcal{T}, T \supseteq \mathcal{TG} \), functions which are defined for \( T \) are also applicable to \( TG, \mathcal{R}, \mathcal{RG}, \mathcal{T}, \mathcal{TG} \), this is in particular true for the functions Cons, Var, Noc, Par, Pos, and the functions for position arithmetic, denoted with the operators .\. and .[]., to be introduced in the rest of this section.
5.2. THE FORMAL DEFINITION

The quintessence of the set of well-formed terms is that immediate descendant constructors that are allowed to occur under a certain parent constructor are restricted by the function $\rho$ and root constructors are restricted by the set $S$. Variables can only match against, and stand for, sub-terms of which the root constructor is member of the $\mu$-value of that variable. In practice it may be necessary to impose extra conditions on the components $\rho$, $\mu$, and $S$ of a certain vocabulary. Below is a presentation of a number of possible properties of retrenched vocabularies.

**Definition 5.9.** A vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ will be called integrated if and only if each constructor $c \in C$ is integrated, and a constructor $c \in C$ is integrated if and only if $c \in S$ or $c \in \rho(c', i)$ for some integrated $c' \in C$ and some $1 \leq i \leq \alpha(c')$.

The set $C_r$ of integrated constructors is easily seen to be the limit of the sequence

- $C_r^0 = S$,
- $C_r^{n+1} = C_r^n \cup \bigcup_{c \in C_r, 1 \leq i \leq \alpha(c)} \rho(c, i)$.

Obviously the sequence $C_r^i$ converges after a finite number of iterations if $C$ is a finite set (which was assumed in definition 5.1). So for finite vocabularies there is a decision procedure for the set $C_r \subseteq C$ and for the fact whether or not the vocabulary is integrated.

**Definition 5.10.** A vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ will be called satisfiable if and only if each constructor $c \in C$ is satisfiable, and a constructor $c$ is satisfiable if and only if $c$ is a constant or $\rho(c, i)$ contains at least one satisfiable constructor for each $1 \leq i \leq \alpha(c)$.

The set $C_s$ of satisfiable constructors is easily seen to be the limit of the sequence:

- $C_s^0 = K$,
- $C_s^{n+1} = C_s^n \cup \{ c \in C \mid \forall 1 \leq i \leq \alpha(c) \exists c' \in C_s^n \ [\rho(c, i) \ni c'] \}$.

Obviously the sequence $C_s^i$ converges after a finite number of iterations if $C$ is a finite set (which was assumed in definition 5.1). So for finite vocabularies there is a decision procedure for the set $C_s \subseteq C$ and for the fact whether or not the vocabulary is integrated.

**Definition 5.11.** A vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ will be called trivial combination free if and only if $\forall yc \in YC \ [|yc| > 1]$.

**Definition 5.12.** A vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ will be called normalized if and only if $\forall y_1, y_2 \in YV \ [y_1 = y_2 \lor y_1 \cap y_2 = \emptyset]$.

**Definition 5.13.** A vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ will be called uniquely starting if and only if $\forall c \in C, \forall 1 \leq i \leq \alpha(c) \ [S \cap \rho(c, i) = \emptyset]$.
A short explanation for each of the five defined properties:

- In a vocabulary which is not integrated some constructors cannot appear in well-formed terms because they are simply allowed neither at root positions nor at non-root positions (below integrated constructors).

- In a vocabulary which is not satisfiable some constructors cannot appear in well-formed terms for a different reason: there are no sub-terms that can be positioned on some of the operand positions of these constructors.

- In a vocabulary which is not trivial combination free some pairs of constructors will always appear together in well-formed (ground) terms. In this situation a transformation which eliminates this trivial combination is very obvious.

- A vocabulary which is normalized corresponds to an order sorted signature with an empty relation on the types, i.e. a many sorted signature. Section 5.4.4 presents a proof.

- The property "uniquely starting" means that there is a strict division between constructors that can appear at root-positions and constructors that can appear at non-root positions. The property is interesting for reasons that will become clear in chapter 8 and 9.

It will appear that vocabularies (and entire systems) that do not possess a certain property, can be transformed into vocabularies that do possess this property, for each of the five defined properties. This is rather easy for integratedness and satisfiability, but less simple for the other properties. Details of these transformations will be presented in chapter 9.

5.3 Term Analysis for Vocabularies

The equivalents of the definitions from section 2.3 will be presented in this section. Most definitions of section 2.3 can be adapted with minor modifications. All adapted definitions are included here for completeness, except the definition of the Type function, because the type notion is not explicitly present in the vocabulary formalism.

Definition 5.14. Let $A = (C, V, S, \alpha, \rho, \mu)$. The function $Cons : T \rightarrow \mathcal{P}(C)$ returns the set of constructors that occur in a given term. A recursive definition:

- $Cons(k) = \{k\}$, if $k \in K$;
- $Cons(v) = \emptyset$, if $v \in V$;
- $Cons(f(t_1, \ldots, t_{\alpha(f)})) = \{f\} \cup \bigcup_{i=1}^{|\alpha(f)|} Cons(t_i)$, if $f \in C - K$. \hfill \Box
5.3. TERM ANALYSIS FOR VOCABULARIES

Definition 5.15. Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$. The function $\text{Var} : T \rightarrow \mathcal{P}(V)$ returns the set of variables that occur in a given term. A recursive definition:

- $\text{Var}(k) = \emptyset$, if $k \in K$;
- $\text{Var}(v) = \{v\}$, if $v \in V$;
- $\text{Var}(f(t_1, \ldots, t_{\alpha(f)})) = \bigcup_{i=1}^{\alpha(f)} \text{Var}(t_i)$, if $f \in C - K$. $\square$

Definition 5.16. Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$. The function $\text{Noc} : (O \uplus V) \times T \rightarrow \mathcal{N}$ returns the number of occurrences of a given symbol in a given term. A recursive definition:

- $\text{Noc}(\text{sym}, t) = 0$, if $t \in (K \cup V)$ and $\text{sym} \neq t$;
- $\text{Noc}(\text{sym}, t) = 1$, if $t \in (K \cup V)$ and $\text{sym} = t$;
- $\text{Noc}(\text{sym}, f(t_1, \ldots, t_{\alpha(f)})) = \sum_{i=1}^{\alpha(f)} \text{Noc}(\text{sym}, t_i)$,
  if $f \in (C - K)$ and $\text{sym} \neq f$;
- $\text{Noc}(\text{sym}, f(t_1, \ldots, t_{\alpha(f)})) = 1 + \sum_{i=1}^{\alpha(f)} \text{Noc}(\text{sym}, t_i)$,
  if $f \in (C - K)$ and $\text{sym} = f$. $\square$

Definition 5.17. Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$. The function $\text{Par} : (C \uplus V) \times T \rightarrow \mathcal{P}(C \times \mathcal{N})$ returns the set of constructor-index pairs under which the given symbol occurs in the given term. A recursive definition:

- $\text{Par}(\text{sym}, v) = \emptyset$, for all $v \in V$;
- $\text{Par}(\text{sym}, k) = \emptyset$, for all $k \in K$;
- $\text{Par}(\text{sym}, f(t_1, \ldots, t_n)) = \{(f, i) \mid \text{Root}(t_i) = \text{sym}\} \cup \bigcup_{i=1}^{n} \text{Par}(\text{sym}, t_i)$,
  for all $f \in C$ with $\alpha(f) = n$. $\square$

Definition 5.18. Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$. A substitution is a total function $\sigma : V \rightarrow \mathcal{R}$, for which:

$\forall v \in V \ [\text{if } \sigma(v) \in V \text{ then } \mu(\sigma(v)) \subseteq \mu(v) \text{ else } \text{Root}(\sigma(v)) \in \mu(v)]$.

The domain of substitutions is extended from variables to terms, $\sigma : T \rightarrow T$, in such a way that:

- $v^\sigma = \sigma(v)$ for all variables $v \in V$;
- $k^\sigma = k$ for all constants $k \in K$;
• \( f(t_1, ..., t_n)^\sigma = f(t_1^\sigma, ..., t_n^\sigma) \) for all \( f \in C \) with \( \alpha(f) = n > 0 \).

Superscript notation will be used for the extended version. Any substitution \( \alpha : V \rightarrow V \) is called a \textit{variable substitution}.

**Definition 5.19.** An \textit{alpha conversion} of a term \( t \) is the result of a reversible, non-identical substitution operation on \( t \), i.e., a term \( t^\sigma \neq t \) with \( \sigma \) such that there is a substitution \( \sigma' \) with \((t^\sigma)^{\sigma'} = t\). An \textit{alpha conversion} of a pair of terms \( \langle l, r \rangle \) (equations, rewrite rules), is the result of a reversible, non-identical substitution operation applied to both components, i.e., a pair \( \langle l^\sigma, r^\sigma \rangle \neq \langle l, r \rangle \) with \( \sigma \) such that there is a substitution \( \sigma' \) with \((\langle l^\sigma, r^\sigma \rangle)^{\sigma'} = \langle l, r \rangle \). For a set \( S \) of terms or pairs of terms \([S]_\alpha\) denotes the largest subset \( S_{sub} \) of \( S \) such that \( S_{sub} \) contains no terms or pairs which are alpha convertible into each other. Note that \([S]_\alpha\) is unique up to systematic renaming of variables.

**Definition 5.20.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). A variable \( v \in V \) is called \textit{maximally typed} in a term \( t \in T \) if and only if there is no variable substitution \( \sigma = \{ v' \leftarrow v \} \) with \( v' \in V \) and \( \mu(v) \subseteq \mu(v') \) (proper inclusion, so \( \mu(v) \neq \mu(v') \)) and a term \( t' \in T \) such that \( t = t'^\sigma \). A variable \( v \) is called \textit{maximally typed} in a pair of terms \( \langle l, r \rangle \in T \times T \) (equations, rewrite rules) if and only if there is no variable substitution \( \sigma = \{ v' \leftarrow v \} \) with \( v' \in V \) and \( \mu(v) \subseteq \mu(v') \) and a pair of terms \( \langle l', r' \rangle \in T \times T \) such that \( \langle l, r \rangle = \langle l'^\sigma, r'^\sigma \rangle \).

**Definition 5.21.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). A \textit{rewrite rule} for vocabulary \( A \) is a pair \( \langle l, r \rangle \) consisting of a left hand side pattern \( l \) and a right hand side pattern \( r \) for which:

1. \( l, r \in \mathcal{R} \),
2. \( l \notin V \),
3. \( \text{Var}(r) \subseteq \text{Var}(l) \).

**Definition 5.22.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). The set of \textit{positions} in a term \( t \in T \), denoted by \( \text{Pos} : T \rightarrow \mathcal{P}(N^*) \), is defined formally as:

- \( \text{Pos}(v) = \{ \varepsilon \} \) for all \( v \in V \),
- \( \text{Pos}(k) = \{ \varepsilon \} \) for all \( k \in K \),
- \( \text{Pos}(f(t_1, ..., t_n)) = \{ i.p \mid 1 \leq i \leq n, p \in \text{Pos}(t_i) \} \) for all \( f \in C - K \) and all \( t_1, ..., t_n \in T \).

A position \( p \in N^* \) is called a \textit{prefix} of \( p' \in N^* \) if there is a string \( s \in N^* \) such that \( p.s = p' \). If \( p \) is a prefix of \( p' \) and \( p \neq p' \) then \( p \) may also be called a \textit{proper prefix} of \( p' \).

**Definition 5.23.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). The \textit{subterm at position} \( p \in \text{Pos}(t) \) in a term \( t \in T \), denoted by \( t|_p \), is defined formally as:
5.3. TERM ANALYSIS FOR VOCABULARIES

• \( t\ls = t \),
• \( op(t_0, \ldots, t_n)\ls = t_0\ls \).

A pattern \( l \in \mathcal{R} \) is said to match in \( t \in \mathcal{T} \) if and only if there is a position \( p \in \text{Pos}(t) \) and a substitution \( \sigma : V \rightarrow \mathcal{T} \) such that \( t\ls = l\sigma \). \( \square \)

**Definition 5.24.** The replacement in \( t \in T \) of a subterm at position \( p \in \text{Pos}(t) \) by a term \( t_{new} \in T \), denoted by \( t[t_{new}]_p \), is defined formally as:

• \( t[t_{new}]_p = t_{new} \),
• \( op(t_0, \ldots, t_n)[t_{new}]_p = op(t_0, \ldots, t_i[t_{new}]_p, \ldots, t_n) \).

**Definition 5.25.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) be a vocabulary and \( h = \langle l, r \rangle \) be a rewrite rule for \( A \). The one step rewrite relation \( \rightarrow_h \subseteq T_A \times T_A \) induced by \( h \) is defined as: \( t \rightarrow_h t' \) if and only if there exists:

• a position \( p \in \text{Pos}(t) \),
• a substitution \( \sigma : V \rightarrow TR, \)

such that: \( t\ls = l\sigma \) and \( t' = t[r\sigma]\ls \). The transitive reflexive closure of \( \rightarrow_X \) is denoted by: \( \rightarrow_X^* \), the reverse relation of \( \rightarrow_X \) by: \( \leftarrow_X \) and the transitive reflexive closure of \( \leftarrow_X \) by: \( \leftarrow_X^* \). The replacement of a subterm \( l\sigma \) by a subterm \( r\sigma \) in a term \( t \), based on rule \( h = \langle l, r \rangle \), is called the firing of rule \( h \). \( \square \)

**Definition 5.26.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). A rewrite rule \( h = \langle l, r \rangle \) for \( A \) is said to be collapsing if and only if \( r \in V \) (if the right hand side of the rule is a single variable). \( \square \)

**Definition 5.27.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). A rewrite rule for vocabulary \( A \) is said to be well-formedness preserving if and only if \( \mathcal{T} \) is closed under application of this rule. \( \square \)

**Theorem 5.28.** Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). A rewrite rule \( h = \langle l, r \rangle \), with \( \text{Root}(l) = c_l \) and \( \text{Root}(r) = c_r \) is well-formedness preserving if and only if the following holds:

• for non-collapsing rules:
  \[ \forall (c, i) \in \rho^{-1}(c_l) \ [c_r \in \rho(c, i) \land c_l \in S \Rightarrow c_r \in S], \]

• for collapsing rules:
  \[ \forall (c, i) \in \rho^{-1}(c_l) \ [\mu(c_r) \subseteq \rho(c, i) \land c_l \in S \Rightarrow \mu(c_r) \subseteq S]. \]
Proof of if part. Well-formedness of terms is related to what constructors occur at which positions of other constructors and to what constructors occur at the root position (definition 5.5 and 5.7). Well-formedness violations (corresponding to type violations in the sorted signature formalism) have actually to do with two constructor positions: a position of a parent and a position of a child. It is related to the edges in a tree representation of the term. The root restriction can be viewed as pertaining to an imaginary edge pointing to the root node. In the rest of this proof these edge positions (or possible edge positions) will be called places. In the condition of the theorem the following cases were treated separately:

1. non-collapsing rules,

2. collapsing rules.

Some more things have to be discriminated between, to decompose the proof in manageable parts. With rewriting a well-formed term based on a rewrite rule the only places at which violations of the restrictions of $\rho$ and $S$ can be introduced are places in and around the location of the subterm to be replaced. Because well-formedness of the rewrite rule is a pre-condition (which implies well-typedness of variables), violations of $\rho$ and $S$ can only be at the place directly above the root constructor of the subterm which is involved in the replacement. Violations of $\rho$ cannot occur inside the replacing subterm or at the places between the subterm and its variables because rules, and in particular right hand sides of rules, are well-formed. The place in the well-formed term to be rewritten at which the root of the left hand side is exchanged for the root of the right hand side is potentially dangerous for the introduction of type violations. Rules can match at:

A. root positions of terms to be rewritten,

B. non-root positions of terms to be rewritten.

In the former case possible violations pertain to $S$, in the latter case possible violations pertain to $\rho$. A left hand side can only match at a root position if the root of the left hand side is in $S$. The component:

$$c_l \in S \Rightarrow c_r \in S$$

ensures therefore that $S$-restrictions are not violated at root places of the rule patterns (case 1.A). A left hand side can only match at a non-root position if the root of the left hand side is in $\rho(c_{pa}, p)$ with $c_{pa}$ the constructor directly above that position and $p$ the particular index value for $c_{pa}$. The component:

$$\forall (c, i) \in \rho^{-1}(c_l) \left[ c_r \in \rho(c, i) \right]$$
ensures therefore that ρ-restrictions are not violated at variable places of the rule patterns (case 1.B). An alternative formulation of this requirement is: \( \rho^{-1}(c_l) \subseteq \rho^{-1}(c_r) \).

The above case analysis was for non-collapsing rewrite rules. Collapsing rules need special attention because they do not have a constructor at the root of the right hand side, nor do they have constructors "above" the variables of the right hand side. Positions and places of root and variables coincide in the right hand side of a collapsing rule. As with non-collapsing rules, collapsing rules can match at root positions of terms to be rewritten and at non-root positions (cases again denoted by A and B). Because of the coincidence between root position and variables position in right hand side rule, possible violations of constructor restrictions pertain to both ρ and S. The constructors that can act as root of a term which is bound to the variable \( v \) which constitutes the right hand side of the rule is given by \( μ(v) \). These constructors will appear at places where the root of left hand side rule is allowed to occur so the component:

\[ c_l \in S \Rightarrow μ(c_r) \subseteq S \]

ensures well-formedness of the result in case of a match at the root position of the term to be rewritten (case 2.A). Note that \( c_r \) is the variable which constitutes the right hand side of the rule. And the component:

\[ \forall (c, i) \in ρ^{-1}(c_l) [μ(c_r, h_l) \subseteq ρ(c, i)] \]

ensures well-formedness of the result in case of a match at a non-root position of the term to be rewritten (case 2.B). It is easy to see that the cases that have been dealt with cover all possible situations. This completes the if-part of the proof.

**Proof of only-if-part.** The cases of non-collapsing rules and collapsing rules will again be treated separately.

Suppose \( h = ⟨l, r⟩ \) is a non-collapsing rewrite rule, i.e. \( r \) is not a single variable. If the first part of the condition for non-collapsing rules doesn’t hold then \( ∃(c, i) \in ρ^{-1}(c_l) [c_r \notin ρ(c, i)] \), suppose that \( c_r \notin ρ(c_x, y) \). Because \( c_l \in ρ(c_x, y) \) it is easy to construct a well-formed term that can be rewritten to a non-well-formed term, based on \( h \). Take for instance:

\[ c_x(v_1, ..., v_{y-1}, l, v_{y+1}, ..., v_{α(c_x)}) \]

with: \( μ(v_i) \subseteq ρ(c_x, i) \) for all \( 1 \leq i \leq α(c_x) \) and \( i \neq y \). If the second part of the condition for non-collapsing rules doesn’t hold then \( c_l \in S \) and \( c_r \notin S \). A well-formed term which can be rewritten to a non-well-formed term, based on \( h \) is for instance: \( l \).

Suppose \( h = ⟨l, r⟩ \) is a collapsing rule, i.e. \( r \) is a variable. If the first part of the condition for non-collapsing rules doesn’t hold then \( ∃(c, i) \in ρ^{-1}(c_l) [μ(c_r) \not\subseteq ρ(c, i)] \), suppose that \( μ(c_r) \not\subseteq ρ(c_x, y) \). Again it is easy to construct a well-formed term that can be rewritten to a non-well-formed term, based on \( h \). Take for instance:
with: $\mu(v_i) \subseteq \rho(c, i)$ for all $1 \leq i \leq \alpha(c)$ and $i \neq y$. The well-formedness violation is in this case caused by the variable $r = c_r$ which has a type too large to fit into position $y$ of $c_z$. If the second part of the condition for collapsing rules doesn't hold then $c_i \in S$ and $\mu(c_r) \not\subseteq S$. A well-formed term which can be rewritten to a non-well-formed term, based on $h$ is for instance: $l$, because $c_r \in V$ has a type which is too large for $S$.

The counter examples indicate that each component of the condition is necessary, which completes the proof. □

**Definition 5.29.** A *retrenched rewrite system* is a pair $(A, RS)$ consisting of a vocabulary $A$ and a set of well-formedness preserving rewrite rules $RS$. The name “retrenched rewrite system” will usually be abbreviated to just “rewrite system”. The one step rewrite relation $\rightarrow_{RS}$ induced by a rewrite system $(A, RS)$ is defined as: $t \rightarrow_{RS} t'$ if and only if there exists a rule $h \in RS$ such that $t \rightarrow_h t'$. □

**Definition 5.30.** An *untyped retrenched vocabulary* is a vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ with:

- $\rho(c, i) = C$ for all $c \in C$ and all $1 \leq i \leq \alpha(c)$,
- $\mu(v) = C$, for all $v \in V$,
- $S = C$.

**Definition 5.31.** An *untyped retrenched rewrite system* is a retrenched rewrite system $(A, RS)$ with an untyped vocabulary $A$. □

**Definition 5.32.** Let $A = (C, V, S, \alpha, \rho, \mu)$. The *inverse restriction function* is a function $\rho^{-1} : C \rightarrow \mathcal{P}(C \times N)$ which returns sets of pairs of constructors and indices. The inverse restriction function is the generalized inverse of the restriction function $\rho$. Formal definition:

$$\rho^{-1}(c) = \{ (c', i) \mid c \in \rho(c', i) \}.$$

**Definition 5.33.** Let $A = (C, V, S, \alpha, \rho, \mu)$. The *inverse match function* is a function $\mu^{-1} : C \rightarrow \mathcal{P}(V)$ which returns sets of variables. The inverse match function is the generalized inverse of the match function: $\mu^{-1} : C \rightarrow \mathcal{P}(V)$. Formal definition:

$$\mu^{-1}(c) = \{ v \in V \mid c \in \mu(v) \}.$$

**Definition 5.34.** The set $FT_A$ of fragments over vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ is defined as the set of terms $T'_N$ over the modified vocabulary:

$$A' = (C \cup \{ \Box \},$$
5.3. TERM ANALYSIS FOR VOCABULARIES

\[ V, \]
\[ \alpha \uparrow \{ \square \leftarrow 0 \}, \]
\[ \{c, m\} \leftarrow \rho(c, m) \cup \{\square \} \mid c \in C, 1 \leq m \leq \alpha(c) \}, \]
\[ \{v \leftarrow \mu(v) \cup \{\square \} \mid v \in V \}, \]
\[ S \cup \{\square \} \]

The sets \( FTG \) of ground fragments, \( FR \) of restricted fragments, \( FRG \) of restricted ground fragments, \( FT \) of well-formed fragments, and \( FTG \) of well-formed ground fragments are defined similarly as \( TG, R, RG, T \), and \( TG \) over the modified vocabulary \( A' \). The special constant "\( \square \)" denotes a hole in the fragment.

\[ \square \]

Definition 5.35. Let \( A \) be a vocabulary and \( FT_A \) the set of fragments over \( A \). The \textit{Rank} of a fragment \( fr \), denoted by \( \text{Rank} : FT \rightarrow N \), is the number of holes in it, formally:

\[ \text{Rank}(fr) = \text{Noc}(\square, fr). \]

\[ \square \]

Definition 5.36. Let \( A \) be a vocabulary and \( FT_A \) the set of fragments over \( A \). The function \( \text{Nunhp} : FT_A \rightarrow N \) returns the number of non-hole positions in a given fragment. A formal definition:

\[ \text{Nunhp}(fr) = | \{ p \in \text{Pos}(fr) \mid fr|_p \neq \square \} |. \]

\[ \square \]

Definition 5.37. The \textit{fragment notation} for a term or fragment is defined as follows:

\[ fr[fr_1, ..., fr_n] \]

denotes the term (or fragment) that results from replacing the \( n \) holes in the rank-\( n \) fragment \( fr \), in left-to-right order, by the \( n \) terms (or fragments) \( fr_i, 1 \leq i \leq n \).

\[ \square \]

Definition 5.38. The set of allowed constructors to act as root of a subterm to be substituted for a hole in a fragment is returned by the \textit{Aset} operator (ALlowed SET).

\[ \text{Aset} : FR \times N \rightarrow \mathcal{P}(C), \]
\[ \text{Aset}(fr, i) = \{ \text{Root}(t) \mid fr[\square_1, ..., \square_{i-1}, t, \square_{i+1}, ..., \square_{\text{Rank}(fr)}] \in FR \}. \]

The indices of the \( \Box \)s are for counting purposes. Each \( \Box_i \) equals \( \Box \): the special constant which represents a "hole".

\[ \square \]
5.4 Relations between Vocabularies and Signatures

5.4.1 Introduction

The relationship between the retrenched vocabulary formalism and the order sorted signature formalism will be explored in this section. The retrenched vocabulary formalism is essentially a stripped down version of the order sorted signature formalism with some extra functionality. The differences between the sorted signature and the vocabulary formalism are:

1. a restriction of the set of constructors which is allowed to act as root constructor of well-formed terms.

2. a shortcut between operand and result types of operators.

The second item is only a superficial, syntactical difference. It is possible to state the theory about transformations that will be presented in chapter 6 in terms of conventional operators with operand types and result types (which will be done partly in chapter 7) but this will turn out to be very awkward, because each transformation will require a major restructuring of the type system. The shortcut of types is only for convenience of the presentation of our theory. The first item represents a structural difference between the conventional and the new formalism. Our theory requires the ability to preclude certain constructors from acting as root constructors of well-formed terms. This functionality can be added to the conventional sorted signature formalism by extending the formalism with a specially dedicated root-type which prescribes the valid result type of root operators of well-formed terms.

It is easy to derive a vocabulary from an order sorted signature and vice versa. The relation between both formalisms will be made explicit by giving a scheme for the derivation of vocabularies from signatures in section 5.4.2, and a scheme for the derivation of signatures from vocabularies in section 5.4.3. Section 5.4.4 is about properties of the described derivation processes.

5.4.2 The Derivation of Vocabularies from Signatures

The derivation of vocabularies from signatures is straightforward. The components \( C, V, \) and \( \alpha \) are directly available as derived quantities of signatures (definition 2.1). The definitions of \( \rho \) and \( \mu \) can be given in terms of the earlier introduced Fits-function (definition 2.3), a function which matches a given type (an operand type or a variable type) with the result types of each of the available constructors. The root restriction set \( S \) can simply be defined to be the entire constructor set, as there are no restrictions for root operators in the conventional order sorted signature formalism. The derivation process is rendered in a formal way in the following scheme.

Scheme 5.39. A retrenched vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) can be derived from an order sorted signature \( \Sigma = \langle S, Rdec, Odec, Vdec \rangle \) as follows. Symbols left of
the equal signs pertain to $A$, symbols right of the equal signs pertain to $\Sigma$, unless specified otherwise:

\[
C = \{ \text{op} \mid \langle \text{op}, \text{typespec} \rangle \in O\text{decs} \} \ (= O_\Sigma),
\]

\[
V = \{ \text{var} \mid \langle \text{var}, \text{type} \rangle \in V\text{decs} \} \ (= V_\Sigma),
\]

\[
S = C_A,
\]

\[
\alpha = \{ \text{op} \leftarrow n \mid \langle \text{op}, \langle s_1, ..., s_n, s_r \rangle \rangle \in O\text{decs} \} \ (= \alpha_\Sigma),
\]

\[
\rho = \{ \langle \text{op}, i \rangle \leftarrow \text{Fits}(O\text{type}(\text{op}, i)) \},
\]

\[
\mu = \{ \text{var} \leftarrow \text{Fits(type)} \mid \langle \text{var}, \text{type} \rangle \in V\text{decs} \}.
\]

It is easy to see that the requirements for vocabularies as stated in definition 5.1 are satisfied for each vocabulary that is derived by scheme 5.39. It is also easy to see that there is a bijection between well-formed terms over $\Sigma$ and well-formed terms over a vocabulary $A$ which is derived from $\Sigma$, a bijection which maps terms to "similar looking terms". It is possible to define this bijection formally and to verify certain things but this is tedious and quite trivial. We count on the reader's intuition.

In spite of the bijection between signature terms and vocabulary terms some information of signatures will get lost in the derivation process of scheme 5.39. There is no explicit type notion in the vocabulary formalism, so a distinction between types which is not reflected in a different set of allowed constructors for some operand position will disappear. Types which do not occur as operand type or as result type of operators will also disappear. The identification of some types and the disappearance of other types will become apparent after a composed derivation in which a signature is derived from a derived vocabulary. The changes in the type system after such a composed derivation are similar to the changes that can be brought about by the "type system transformations" of section 7.4.

### 5.4.3 The Derivation of Signatures from Vocabularies

Types in the order sorted signature formalism correspond roughly to sets of constructors in the retrenched vocabulary formalism. These sets of constructors appear as function values of the function $\rho$ and $\mu$, and as root restriction set $S$ (definition 5.1). The sets of constructors that appear in a certain vocabulary are explicitly available in the set $Y$ (definition 5.1). In order to derive an order sorted signature from a vocabulary it is not sufficient to associate a type with each set in $Y$ because overlapping constructor sets are allowed, and with one type for each set it is not clear what the result types of the constructors in the overlap should be. An example to illustrate this.

**Example 5.40.** The derivation of types from constructor subsets. Consider a vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:
- \( C \supseteq \{f, g, h\} \),
- \( Y \supseteq \{\{f, g\}, \{g, h\}\} \).

Suppose that we have the types \( \{f, g\} \) and \( \{g, h\} \) at our disposal in the signature which is supposed to define a similar term set as the vocabulary \( A \). Suppose moreover that we have assigned the type \( \{f, g\} \) to the operand positions where \( f \) and \( g \) are allowed to occur, and the type \( \{g, h\} \) to the operand positions where \( g \) and \( h \) are allowed to occur. In this situation it is clear what the result types of \( f \) and \( h \) should be. However, there is a problem with \( g \): this operator has to be allowed at both categories of operand positions and variables. The only way to accomplish this is to introduce a new type \( \{g\} \), corresponding to the overlap between \( \{f, g\} \) and \( \{g, h\} \), and which is defined to be a subtype of both \( \{f, g\} \) and \( \{g, h\} \).

This argument for two overlapping constructor sets holds in a similar way for three or more overlapping constructor sets. Consider a vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

- \( C \supseteq \{f_1, f_2, f_3, g_1, g_2, g_3, h\} \),
- \( Y \supseteq \{\{f_1, g_1, g_2, h\}, \{f_2, g_2, g_3\}, \{f_3, g_3, g_1, h\}\} \).

In order to assign the right result types to the involved constructors there should be types associated with the sets \( \{g_1, g_2, h\}, \{g_2, g_3, h\}, \{g_3, g_1, h\} \), and \( \{h\} \) in addition to the sets of \( Y \). A graphical illustration of the situation:

![Figure 5.1: correspondence between constructor sets and types](image)

Besides types associated with each possible distinct intersection of one, two or more elements of \( Y \) ("one" being the trivial intersection: the set itself), there
should be a general rest type for constructors which do not occur in any element of \( Y \). These constructors cannot occur in well-formed terms and do not seem very useful therefore. However, neither the vocabulary nor the signature formalism forbids constructors of this kind in general; this is why the derivation process has to support them. The observations of this section are condensed and rendered formally in the following scheme.

**Scheme 5.41.** An order sorted signature \( \Sigma = (S, R\text{decs}, O\text{decs}, V\text{decs}) \) can be derived from a retrenched vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) as follows. Symbols left of the equal signs pertain to \( \Sigma \), symbols right of the equal signs pertain to \( A \), unless specified otherwise.

\[
S = \{ \overline{\bigcap Y} \mid Y \subseteq Y, Y \neq \emptyset, \bigcap Y \neq \emptyset \} \cup \\
\{ C - \bigcup Y \}, \\
R\text{decs} = \{ (\overline{C_l}, \overline{C_r}) \mid \overline{C_l}, \overline{C_r} \in S_\Sigma, C_l \subseteq C_r \}^-,
\]

\[
O\text{decs} = \{ (\langle op, \langle s_1, \ldots, s_n, s_r \rangle \rangle \rangle \mid \alpha(op) = n, s_1, \ldots, s_n \in S_\Sigma, s_r \in S_\Sigma, \\
\forall 1 \leq i \leq n \ [s_i = \overline{\rho(op, i)}], \]

\[
s_r = \text{if } op \in \bigcup Y \text{ then } \{ y \in Y \mid op \in y \} \text{ else } C - \bigcup Y, \]

\[
V\text{decs} = \{ \langle \text{var}, \text{type} \rangle \in V \times S \mid \overline{\mu(\text{var})} = \text{type} \},
\]
in which the overbar (over \( C_\text{a} \) for instance) denotes an operator that hides the set structure of its argument (the only property this operator has to have is: \( \forall S_1, S_2 \subseteq C \ [S_1 = S_2 \iff \overline{S_1} = \overline{S_2}] \)). The superscript "\(^{-}\)" operator in the definition of \( R\text{decs} \) was introduced in section 2.2.

It is easy to see that the requirements for signatures as stated in definition 2.1 are satisfied for each signature that is derived by scheme 5.41 if there is no restriction for root constructors in the vocabulary, i.e. if \( S = C \). It is also easy to see that there is a bijection between well-formed terms over \( \Sigma \) and well-formed terms over a vocabulary \( A \) which is derived from \( \Sigma \), a bijection which maps terms to "similar looking terms". It is possible to define this bijection formally and to verify certain things but this is tedious and quite trivial. We count on the reader's intuition. \( \square \)

### 5.4.4 Properties of Vocabularies Derived from Signatures

In this subsection some properties related to the correspondence between vocabularies and signatures will be presented in a formal way. The properties, pertaining to many-sortedness of signatures, will be used in chapter 9 and 10.

**Theorem 5.42.** A vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) which is derived from a many sorted signature \( \Sigma = (S, R\text{decs}, O\text{decs}, V\text{decs}) \) with scheme 5.39 is normalized in the sense of definition 5.12.
Proof: according to the definition of the normalized notion (definition 5.12) and the sets $YC$ and $YY$ (definition 5.1) we have to show that the function values of $\rho$ and $\mu$ of the derived vocabulary are either identical or non-overlapping. In the construction of the function values of $\rho$ and $\mu$ in scheme 5.39 can be seen that each of these function values is a set:

$$\{op \in O | \langle op, \langle s_1 ... s_n, s_r \rangle \rangle \in Odecs, s_r = s\}$$

for some $s \in S$, since many sorted signatures have an empty $\prec$. The fact to be proven is now obvious. □

Theorem 5.43. A signature $\Sigma = \langle S, Rdecs, Odecs, Vdecs \rangle$ which is derived from a normalized vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with scheme 5.41 is a many sorted signature.

Proof: according to the definition of many-sortedness (definition 2.4) we have to show that $\prec$ is empty. This is of course the case if $Rdecs$ is empty. In the construction of $S$ in scheme 5.41 can be seen that elements of $S$ correspond to function values of $\rho$ and $\mu$. These function values (sets) do not overlap each other properly (some of them may be equal however) because the vocabulary is normalized. The consequence of this is that $Rdecs$ will remain empty, as can be seen in the definition of $Rdecs$. □

5.5 Relations between Vocabularies and Grammars

5.5.1 Introduction

The relationship between the retrenched vocabulary formalism and the context free grammar formalism will be explored in this section. Retrenched vocabularies as defined in section 5.2 can be shown to be equally powerful as context free grammars to define sets of tree-like structures and there is a relatively simple mapping between the two formalisms. Together with the equivalence of retrenched vocabularies and order sorted signatures, which was subject of section 5.4, this means that there is a close relationship between sorted signatures and context free grammars. This relationship is elaborated upon in a practical way in [28] and retrenched vocabularies are in fact an intermediate form between these two formalisms.

The relation between the two formalisms is such that vocabularies can be used to describe context free grammars and vice-versa. The mapping is not difficult but some aspects need justification and explanation. The description of the equivalence between vocabularies and grammars is based on the following conception of context free grammars (taken from [2]):

Definition 5.44. A context free grammar is a quadruple $(S, N, T, P)$ with:

- $S$: the start symbol,
- $N$: the set of non-terminals,
5.5. RELATIONS BETWEEN VOCABULARIES AND GRAMMARS

- $T$: the set of terminals,
- $P$: the set of productions,

for which the following holds:

1. $S \not\in N, S \not\in T,$
2. $N \cap T = \emptyset,$
3. $P \subseteq (N \cup \{S\}) \times (N \cup T)^*.$

Each production consists of a non-terminal and a string of grammar symbols: non-terminals or terminals. Productions will be denoted by pairs consisting of a left hand side non-terminal and a right hand side symbol string. $S$ is in fact also a non-terminal, be it a special one: $S$ is not allowed to occur in right hand sides of productions. Typographic conventions for the rest of this section: non-terminals will be denoted by slanted font upper-case letters, terminals by bold font lower-case letters, and term constructors by slanted font lower-case letters.

5.5.2 The Conversion of Grammars to Vocabularies

To each derivation of a string of terminal symbols from the start symbol of the grammar corresponds a parse tree. Literature in which the relation between terminal strings, context free grammars and parse trees is described is abundant [2, 28]. We will concentrate on the relation between parse trees and terms. Parse trees can be shown to be equivalent with ground terms described by a vocabulary. There is no equivalent of the concept of "variables" in context free grammars. Variables will therefore not be taken in consideration in the conversion processes described in the rest of this section.

Parse trees can be mapped to ground terms by associating unique constructors to productions and unique constants to terminal symbols. The seemingly simplest way of establishing the correspondence: associating constructors with non-terminal symbols and constants with terminals, is not right for a number of reasons. One is that right hand sides of productions with one specific left hand side are are permitted to have a variable length. This would violate the restriction that constructors must have an unambiguously defined arity. Another reason is that when associating constructors with non-terminals the sets of allowed constructors at different positions of one particular constructor may be dependent on each other, i.e. not all combinations of elements from two "allowed sets" of two different positions are valid. An example to illustrate both problems.

Example 5.45. Correspondence between parse trees and ground terms. Consider the following context free grammar $(S, N, T, P)$ with:

$N = \{A, B, C\},$
$T = \{a, b, c, d, e\},$

$T = \{a, b, c, d, e\},$

$P = \{ S \::= bAAb \}$
$A \::= aBBbc$
$A \::= aBbcC$
$A \::= aCbCc$
$A \::= dCd$
$B \::= A$
$B \::= \varepsilon$
$C \::= bb \}.$

If constructors are associated with non-terminals and constants with terminals (take for instance the association $S \equiv s, A \equiv f, B \equiv g, C \equiv h, a \equiv a$ etc.) then the following parse tree does not correspond to a correct ground term:

```
figure 5.2: parse tree
```

since in:

$$s(b, f(a, g, b, g, c), f(d, h(b, b), d), e)$$

the occurrences of constructor $f$ have two different arities. It is possible to transform any context free grammar with non-uniform length right hand side productions into one that does have uniform length right hand side productions and that generates the same set of terminal strings (by renaming some of the conflicting non-terminals and replace them by two or more different non-terminals) but the second problem will always remain. This problem can also be illustrated with the grammar of this example.

When associating constructors with non-terminals (or at least with left hand side non-terminals of productions that have uniform length right hand sides) it is not
always possible to encode the structure of the grammar in the $\rho$-function. Consider
the non-terminal $A$ and the first three productions for $A$. If we associate constructor
$f$ with $A$ then it is obvious what the restrictions of $\rho$ for the first, third and fifth
argument positions should be. At the first position only the constant $a$ is allowed to
occur, etc. The second and fourth positions pose problems however. The constructor
corresponding to non-terminal $C$ is allowed to occur on position two but only if this
constructor does not occur on position four. These dependencies cannot be rendered
by the $\rho$-function. The problems do not occur when associating constructors with
grammar productions. To demonstrate this the $\rho$-function will be defined for this
example. The start set $S$ will be defined as the constructors associated with start
symbol productions (one in this case). The associations of constructors with termin-
als and productions are mirrored in the following extended grammar description:

$$
N = \{ A, B, C \},
$$

$$
T = \{ a \ (< a), b \ (< b), c \ (< c), d \ (< d), e \ (< e) \},
$$

$$
P = \{ S ::= A \ (< s) \\
A ::= aBbBc \ (< f_1) \\
A ::= aBbCc \ (< f_2) \\
A ::= aCbcBc \ (< f_3) \\
A ::= dCd \ (< f_4) \\
B ::= A \ (< g_1) \\
B ::= e \ (< g_2) \\
C ::= bb \ (< h_1) \}. 
$$

The $\rho$-function of a derived vocabulary has to be defined now as:

$$
\rho(s, 1) = \{ f_1, f_2, f_3, f_4 \},
$$

$$
\rho(f_1, 1) = \{ a \}, \rho(f_1, 2) = \{ g_1, g_2 \}, \rho(f_1, 3) = \{ b \};
\rho(f_1, 4) = \{ g_1, g_2 \}, \rho(f_1, 5) = \{ c \},
\rho(f_2, 1) = \{ a \}, \rho(f_2, 2) = \{ g_1, g_2 \}, \rho(f_2, 3) = \{ b \};
\rho(f_2, 4) = \{ h_1 \}, \rho(f_2, 5) = \{ c \},
\rho(f_3, 1) = \{ a \}, \rho(f_3, 2) = \{ h_1 \}, \rho(f_3, 3) = \{ b \};
\rho(f_3, 4) = \{ g_1, g_2 \}, \rho(f_3, 5) = \{ c \},
\rho(f_4, 1) = \{ d \}, \rho(f_4, 2) = \{ h_1 \}, \rho(f_4, 3) = \{ d \},
$$
\[ \rho(g_1, 1) = \{ f_1, f_2, f_3, f_4 \}, \]
\[ \rho(g_2, i) = \bot \text{ for } i = 1 \text{ and } i = 2, \]
\[ \rho(h_1, 1) = \{ b \}, \]
\[ \rho(h_2, 1) = \{ b \}, \]

moreover:
\[ S = \{ s \}. \]

The start set \( S \) of vocabularies that result from the conversion of context free grammars by the process as described above always consists of constructors that do not occur in function values of the restriction function \( \rho \), i.e. the derived vocabularies are uniquely starting (definition 5.13). This can be explained easily from the definition of context free grammars (definition 5.44). Vocabularies that result from the conversion of context free grammars by the process as described above can also be shown to be normalized:

\[ \forall y_1, y_2 \in Y \ [ y_1 = y_2 \lor y_1 \cap y_2 = \emptyset]. \]

A quick glance at the last example is enough to see that it is at least true for this example grammar. This fact is a serious reason to doubt about the possibility to describe arbitrary vocabularies with context-free grammars. However, every vocabulary can be converted into a context free grammar and the result of the converse transformation is always normalized. The reason of this is that not always the most concise vocabulary is produced by the described conversion process. This will be illustrated in example 5.47. We will show in section 5.5.3 that arbitrary vocabularies can be described by context free grammars and in chapter 9 that non-normalized vocabularies can be transformed into equivalent normalized ones.

The derivation as described above will be defined formally. Scheme 5.46 describes the conversion of context free grammars to vocabularies. Both formalisms do not entirely cover each others functionality. Deficiencies in the conversion processes will be discussed. The overbar operator is again an operator to construct new objects out of old ones for which:

\[ \text{object}_1 = \text{object}_2 \iff \overline{\text{object}_1} = \overline{\text{object}_2}. \]

The necessity of the overbar operator is questionable. Its merit is that internal structuring of tuples are hidden and cannot be a source of confusion.

**Scheme 5.46.** A vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) can be derived from a context free grammar \( G = (S, N, T, P) \) as follows (symbols left of the equal signs pertain to \( A \), symbols right of the equal signs pertain to \( G \), unless specified otherwise):

- \( C = \{ \overline{t} \mid t \in T \} \cup \{ \overline{p} \mid p \in P \} \),
• $V, \mu$: not relevant,

• $S = \{ \langle n, \omega \rangle \in C_A \mid n = S \}$,

• $\alpha = \{ \overline{t} \leftarrow 0 \mid t \in T \} \cup \{ \langle n, \omega \rangle \leftarrow |\omega| \mid \langle n, \omega \rangle \in P \}$,

• $\rho = \{ \langle \langle n, s_1 \ldots s_m \rangle, i \rangle \leftarrow \langle \langle n', \omega' \rangle \in C_A \mid n' = s_i \} \cup \{ \overline{t} \in C_A \mid t = s_i \} \mid 1 \leq i \leq m \land \langle n, s_1 \ldots s_m \rangle \in P \}$.

It is easy to see that the conditions for entrenched vocabularies as stated in definition 5.1 hold for vocabularies derived by scheme 5.46. Only the first two conditions need to be verified because the other conditions pertain to variables, which were not taken in consideration. It is also easy to see that there is a bijection between well-formed ground terms over $\Sigma$ and parse trees generated by a grammar $\langle S, N, T, P \rangle$ which is derived from $\Sigma$, a bijection which maps terms to "similar looking parse trees". It is possible to define this bijection formally and to verify certain things but this is tedious and quite trivial. We count on the readers intuition.

A deficiency in the derivation is the identification of $\varepsilon$-productions and terminals. In the rule for the derivation of $\alpha$ can be seen that both terminals and $\varepsilon$-productions are mapped to constants. This is because both terminals and $\varepsilon$-productions represent leaf positions in parse trees. However, there is a difference between terminals and $\varepsilon$-productions. In the "yield" of the parse tree: the concatenation of all leaves from left to right, the $\varepsilon$-productions play no role because they represent the empty string. Terminals play a crucial role in the formation of the yield. If this is a problem the solution can be to furnish each constant with an attribute "textual representation", which is equal to the textual representation of the terminal in case of a terminal and to the empty string in case of an $\varepsilon$-production. The real problem is in fact that the notion "yield" is not supported by the entrenched vocabulary formalism. When deriving a context free grammar from a vocabulary (next scheme) terminals are simply constructed out of constants. It is not possible to define terminals that consist of strings of symbols, like keywords of a programming language for instance. Associating text attributes with constants is therefore in general a good solution for the representation of a formalism in which it is customary to identify terminals with their textual representation.

### 5.5.3 The Conversion of Vocabularies to Grammars

The inverse process of constructing a context free grammar from a normalized vocabulary is straightforward. Associate a non-terminal with each subset of $C$ that occurs as value of the $\rho$-function and reserve an extra symbol to act as start symbol. Associate a terminal with each constant and associate a production with each constructor that occurs somewhere in the set values of the $\rho$-function. The right hand sides of the productions can simply be read from the $\rho$-function. An example that demonstrates that vocabularies need not to be normalized to be convertible into context free grammars:
**Example 5.47.** The conversion of non-normalized vocabularies to context free grammars. Consider the following vocabulary: \((C, V, S, \alpha, \rho, \mu)\) with:

\[
C = \{a, b, c, f_1, f_2, g_1, g_2, g_3, s\};
\]

\(V, \mu: \) not relevant;

\(S = \{s\},\)

\(\alpha(\{a, b, c\}) = \{0\}, \alpha(\{f_1, f_2\}) = \{2\}, \alpha(\{g_1, g_2, g_3, s\}) = \{1\};\)

\(\rho(s, 1) = \{f_1, f_2\},\)

\(\rho(f_1, 1) = \{g_1, g_2\}, \rho(f_1, 2) = \{g_2, g_3\},\)

\(\rho(f_2, 1) = \{g_1, g_2\}, \rho(f_2, 2) = \{a\},\)

\(\rho(g_1, 1) = \{b\},\)

\(\rho(g_2, 1) = \{f_1, f_2\},\)

\(\rho(g_3, 1) = \{c\}.\)

Note that the \(\rho\)-values \(\{g_1, g_2\}\) and \(\{g_2, g_3\}\) overlap. Associating a unique non-terminal with each distinct \(\rho\)-value and reserving a special start non-terminal for the constructor \(s\), leads to the following grammar:

\[(A \equiv \{f_1, f_2\}, B \equiv \{g_1, g_2\}, C \equiv \{g_2, g_3\}, etc.)\]

\[S ::= A\]

\[A ::= BC \mid Ba\]

\[B ::= b \mid A\]

\[C ::= A \mid c\]

Transforming this context free grammar back to a vocabulary will lead to one additional unary constructor. Instead of one constructor \(g_2\) we get two different ones now: name them \(g_{21}\) and \(g_{22}\). The corresponding \(\rho\)-values are:

\(\rho(g_{21}, 1) = \{f_1, f_2\},\)

\(\rho(g_{22}, 1) = \{f_1, f_2\}.\)

The fact the the \(\rho\)-values are equal will be no surprise. It is caused by the fact that the grammar is transformed into a vocabulary with a slightly too general scheme. The two productions:

\[B ::= A,\]
with identical right hand sides allow for more efficient transformations. □

**Scheme 5.48.** A context free grammar \( G = (S, N, T, P) \) can be derived from a vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) as follows (symbols left of the equal signs pertain to \( G \), symbols right of the equal sign pertain to \( A \), unless specified otherwise):

- \( S = \overline{S} \),
- \( N = \{ \rho(c, i) \mid c \in C \land 1 \leq i \leq \alpha(c) \} \),
- \( T = \{ e \mid c \in K \} \),
- \( P = \{ (\overline{C_{s_0}}, \overline{C_{s_1}}, \ldots \overline{C_{s_n}}) \mid (\overline{C_{s_0}} \in N_G \lor \overline{C_{s_0}} = S_G) \land \exists f \in C - K \ [\alpha(f) = n \land f \in C_{s_0} \land \forall 1 \leq i \leq n \ [\rho(f, i) = C_{s_i}]] \}. \)

It is easy to see that the conditions for context free grammars as stated in definition 5.44 hold for grammars derived by scheme 5.48 if the source vocabulary is uniquely starting (definition 5.13). It is also easy to see that there is a bijection between parse trees generated by a grammar \( (S, N, T, P) \) and terms over a vocabulary \( A \) which is derived from \( (S, N, T, P) \), a bijection which maps parse trees to “similar looking terms”. It is possible to define this bijection formally and to verify certain things but this is tedious and quite trivial. We count on the readers intuition.

Deficiencies in the derivation are the special role of the start symbol in the context free grammar formalism and the possibility of empty set \( \rho \)-values in the retrenched vocabulary formalism. If the definition of start symbol is as strict as in definition 5.44 (note that many alternative definitions of the context free grammar formalism exist) then the vocabulary should be required to be “uniquely starting” (definition 5.13). Empty set \( \rho \)-values generate non-terminals (elements of \( N \)) that occur in right hand sides of productions but not in left hand sides: the non-terminals remain undefined. If this is a problem there are at least two solutions possible: simply prohibit empty set \( \rho \)-values (the vocabulary is required to be satisfiable, definition 5.10). Another solution is to define the non-terminals by adding \( \varepsilon \)-productions.
CHAPTER 5. RETRENCHED VOCABULARIES
Chapter 6

Elementary Transformations

6.1 Introduction

Eight categories of simple ground step simulating transformations (definition 3.16) will be introduced in this chapter. The transformations accomplish a minor change in the way terms are represented. The transformations are in a certain sense elementary and more complex transformations can be composed based upon them. The composability of transformations and simulations was treated in chapter 3. Chapter 7 deals with the relation of the various transformations with algebraic semantics. Chapter 8 is devoted to the question whether the name “elementary” is justified and how big the class of transformations is, which can be build with these “elements”. Particular ways to compose transformations to optimize a certain criterion will be discussed in chapter 9.

In section 6.2 a general verbose description will be given of the eight transformations. In section 6.3 a thorough introduction will be presented of how to read and interpret the formal descriptions of the eight transformations, which will be given in section 6.4 through 6.11. The structuring of each of these eight sections is identical: a description of assumptions and eventually some pre-conditions that have to hold for the transformation to be applicable, a description of how to derive a new vocabulary from a certain given one, a definition of a domain mapping, and a description of how to transform a rewrite system for the old vocabulary into one for the new vocabulary. In section 6.12 a general proof of simple ground step simulation is presented for the transformations of section 6.4 through 6.11.

6.2 Informal Description of Transformations

6.2.1 Introduction

The eight categories of elementary transformations will be subdivided in two classes: two structure transformations and six identity transformations. Structure transformations accomplish a modification in the “physical structure” of the terms:
the structure of a term when viewed as graph (tree). Identity transformations accomplish a modification in the identity of the constructors which constitute the term. When viewing terms as graphs (trees) identity transformations accomplish a modification in the labels associated with the nodes of the graph. The two structure transformations are counterparts of each other (they are each others inverses) and the six identity transformations are three pairs of counterparts.

6.2.2 Identity Transformations

It will turn out that various different identity transformations are possible, and possible in two directions. The number of allowed constructors at a certain position can, depending on the context in which the constructor is allowed to occur, be increased (by means of distinguishing different occurrences of one constructor) or decreased (by means of identifying two or more constructors). This distinguishing and identification of constructors can be done in connection with the leaf side context and in connection with the root side context.

The leaf side context of a constructor is always present or always absent: constructors with arity greater than zero always have a leaf side context and constants never have a leaf side context. The root side context is in that respect not as straightforward as the leaf side context. If a constructor is allowed at root positions of terms (if this constructor is member of the start set $S$) then this constructor can have a special "empty" root side context. To overcome the problems with this special context two cases are distinguished, one referred to as the root position case and one referred to as the root side case (from which the root position possibility is precluded). So there are six possible combinations of on the one hand distinguishing or identification and on the other hand leaf side, root side, or root position.

The leaf side context of a constructor $c$ is constituted by the subsets of constructors that are allowed at the various positions of $c$. Constants obviously have no leaf side context. The root side context of a constructor $c$ is constituted by the set of constructor-position pairs at which $c$ is allowed to occur, and the fact whether or not $c$ is allowed to act as root constructor. The leaf side context is directly available in the restriction function $\rho$. The root side context is directly available in the inverse restriction function $\rho^{-1}$, which is defined in terms of $\rho$ (definition 5.32), and in the root restriction set $S$.

Only the six combinations of leaf side, root side, and root position together with distinguishing and identification will be considered in detail. It appears that these six combinations can be considered as elementary and that other kinds of distinguishing and identification can be composed based on these six elementary combinations. Details of this will be given in chapter 8.

Consider a vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with $f \in C$ and a position $p$ with $1 \leq p \leq \alpha(f)$. The leaf side context of $f$ at $p$ is given by $\rho(f, p)$. The root side context of $f$ is given by $\rho^{-1}(f)$ and the root restriction set $S$. Assume:

$$\rho(f, p) = \{c_{h_1}, ..., c_{h_m}\},$$
\[\rho^{-1}(f) = \{\langle p_{a_1}, p_1 \rangle, \ldots, \langle p_{a_n}, p_n \rangle \},\]

\[S = \{r_1, \ldots, r_q \}.

Leaf side distinguishing \(f\) at position \(p\) will be defined as replacing \(f\) by two new constructors, say \(f_1\) and \(f_2\), in \(C\), and replacing \(f\) in terms by \(f_1\) or \(f_2\), depending on what constructor appears at some specific operand position \(p\) of \(f\): part of the leaf side context of \(f\). Leaf side distinguishing of \(f\) into \(f_1\) and \(f_2\) reflects the leaf side context at position \(p\) of \(f\): the identity of the new constructor tells something about the identity of the immediate descendant at position \(p\). Leaf side distinguishing is therefore based on a bi-partition of the set \(\{c_{h_1}, \ldots, c_{h_m} \}\) of allowed descendants at position \(p\).

Root side distinguishing \(f\) will be defined as replacing \(f\) by two new constructors, say \(f_1\) and \(f_2\), in \(C\), and replacing \(f\) in terms by \(f_1\) or \(f_2\), depending on below what constructor and what position of that constructor \(f\) occurs: the root side context of \(f\). Root side distinguishing of \(f\) into \(f_1\) and \(f_2\) reflects the root side context: the identity of the new constructor tells something about the immediate ancestor. Root side distinguishing is therefore based on a bi-partition of the set \(\{\langle p_{a_1}, p_1 \rangle, \ldots, \langle p_{a_n}, p_n \rangle \}\) of possible ancestor-position pairs. Root side distinguishing is not allowed if \(f \in S\).

Root position distinguishing \(f\) will be defined as replacing \(f\) by two new constructors, say \(f_1\) and \(f_2\) in \(C\), and replacing \(f\) in terms by \(f_1\) or \(f_2\), depending on if \(f\) occurs at the root position of the term or at an internal position of the term. Root position distinguishing of \(f\) into \(f_1\) and \(f_2\) reflects the root side context in a way different from root side distinguishing: the identity of the new constructor tells whether the constructor is a root constructor or an internal constructor of the particular term. Because root position distinguishing is in some cases only possible if two or more constructors are distinguished simultaneously, a generalized scheme for root position distinguishing sets of constructors will be presented.

Identification of constructors is just the inverse of distinguishing of constructors. Leaf side identification is the renaming of two different constructors with one new name and the leaf side context as decision criterion for the origin of the new constructor. Root side identification is the renaming of two different constructors with one new name and the root side context as decision criterion for the origin of the new constructor. Root position identification is the renaming of two different constructors with one new name and the position (root or non-root) as decision criterion for the origin of the new constructor. The pre-conditions for constructor identification are more stringent than that of constructor distinguishing. Identification is only possible if the context of the constructors to be identified contains enough information to determine the origin of the new constructor. The constructors to be identified must also have an equal arity.
6.2.3 Structure Transformations

The two transformations that will be described informally in this section accomplish a modification in the structure of terms. The contraction transformation will contract (take together) two vertically related nodes in a term representation. The situation around a contraction transformation can in general be depicted as (the constructor names abbreviate ANcestor, PARENT, CChild, Constructor):

![Figure 6.1: Contraction Transformation](image)

The inverse of the contraction transformation is the expansion transformation. The principle of this transformation is that one constructor will be replaced by two vertically connected constructors with one part of the descendants of the original constructor at the remaining positions of the topmost new constructor and the other part of the descendants of the original constructor at the positions of the bottommost new constructor. Graphically:

![Figure 6.2: Expansion Transformation](image)

6.3 Formal Descriptions of Transformations

6.3.1 Introduction

In the eight ensuing sections for each of the eight elementary transformations there will be given:

- some assumptions and, when necessary, a pre-condition for the transformation to be legitimate,

- formal expressions for the derivation of a new vocabulary from the original one,
6.3. FORMAL DESCRIPTIONS OF TRANSFORMATIONS

- a description of the bijective domain mapping in question by means of a term rewriting system,
- formal descriptions of how to derive a step simulating rewrite system for the new vocabulary from one for the original vocabulary.

6.3.2 Assumptions and Pre-conditions

The pre-conditions for the transformation to be legitimate will be explained when necessary. The following terminology will be used to express the pre-condition for the transformation distinguishing and identification in connection with root position. A constructor $f$ is called:

- **root-specific** if $f \in S \land p^{-1}(f) = \emptyset$,
- **non-root-specific** if $f \notin S \land p^{-1}(f) \neq \emptyset$,
- **non-specific** if $f \in S \land p^{-1}(f) \neq \emptyset$,
- **specific** if it is root-specific or non-root-specific.

6.3.3 Derivation of New Vocabulary

The components of the newly derived (or transformed) vocabulary will be indicated with two primes, so $A'' = (C'', V'', S'', \alpha'', \rho'', \mu'')$. Elements with single primes will be used for dividing the entire expression in easier to understand parts. Sometimes the expression would become to large to be well comprehensible when stated as a whole. In that case components of $A''$ will be defined in terms of components of $A'$ and components of $A'$ in terms of components of $A$. If the expression is relatively simple, the components of $A''$ will be defined directly in terms of the components of $A$. The formal expression for the derivation of new vocabularies will use extensively the notations for the construction and modification of functions as defined in chapter 1.

The definition of the new variable set $V''$ of the new vocabulary needs some explanation. We are deriving vocabularies to support simple ground step simulation, so the terms to be rewritten will not contain variables. The strategy therefore is to define $V''$ such that $V''$ contains enough variables to make the transformation of rewrite systems consisting of rules with variables possible. Because we use variables only in rules, the particular identity of variables is irrelevant (we can always systematically rename variables of rules without altering the functionality of the rule). Therefore we will only prescribe what represented types must be supported by the set of variables $V$ and the $\mu$-function. The definition of retrenched vocabularies (definition 5.1) requires that there be countably infinite variables associated with each represented type, so if the set of represented types is known, $V''$ and $\mu''$ are fixed up to renaming. The set of represented types in the variables was a derived quantity and named $YV$ in definition 5.1. We will therefore give a derivation of $YV''$ from
$YV$ instead of a derivation of $V''$ and $\mu''$ from $V$ and $\mu$ in the derivation schemes for vocabularies.

6.3.4 Definition of Domain Mapping

The description of the domain mapping will be given by a *recursive definition* for the two root position transformations and by an *untyped rewrite system* (definition 5.31) for the other transformations. The reason for the special treatment for root position transformations is that it is not possible to define the domain mappings with a rewrite system: rules are not able to "see" the difference between a match at a root position or at an internal position of a term. The reason why the rewrite systems for the domain mappings have to be untyped is the following. Recall that both domains are typed, or have restrictions on the formation of terms which correspond to types (section 5.4). If the domains are different (which needs not necessarily be the case) then the type-related restrictions will also be different. A domain mapping can therefore only conveniently be described by an untyped rewrite system over a constructor set which is the union of the constructor sets of both domains.

A sufficient condition for a rewrite system to be a function on a set of terms is that it be terminating (strongly normalizing) and confluent. We will adopt that condition. Each of the rewrite implementations that will be presented can easily be seen to be confluent, terminating, and correct with regard to the required functionality, i.e. it implements a domain mapping between the original and the derived vocabulary. Actually each rewrite system that will be presented as a "definition of domain mapping" is also an implementation of $\phi^{-1}$ ($\phi^{-1}$ is required in concrete transformation processes, section 3.1), by reversing the direction of the rules. The $[\cdot]_\alpha$-operator in the expressions for the definition of the rewrite systems is to remove alpha variant rules, which are generated by the set comprehension.

6.3.5 Derivation of Rewrite System

For each rewrite system a new rewrite system has to be derived which covers the functionality of the original system in the new term domain. It appears to be possible to perform this derivation on a rule-by-rule basis, i.e. the derivation of an entire rewrite system consisting of one or more rules can be done by deriving simulating rule sets for each of the individual rules and by taking the union of these single-rule-simulating rule sets. In other words: when performing the derivation of sets of rules, the eventual interactions between individual rules do not have to be taken into account. Sometimes however, it is possible to replace a set of simulating rules which originate from two or more original rules with one more general rule. Generalizations of this kind are described in chapter 9. Note that these generalizations are not necessary in the descriptions of derivations in this chapter: the derived rewrite systems operate correctly before as well as after the generalizations to be presented in chapter 9. The merit of generalizations is only that the number of rules and also the complexity of the patterns is reduced.
Some remarks will be given about the contents of the derivation processes. The nature of the transformations (distinguishings, identifications, contractions, expansions) implies that something has to be done if the involved constructors (the constructors to be distinguished, contracted, etc.) appear in the patterns of the rule or if they can appear in the direct context of a match (an occurrence) of a pattern in a term to be rewritten. Occurrences of involved constructors inside rule patterns will appear to be easily handled; occurrences of involved constructors at the bounds of pattern occurrences in terms to be rewritten (which means: just inside or just outside the pattern occurrence) will sometimes give rise to rule multiplications: the simulation of the functionality of the one original rule needs two or more rules. The algorithmic descriptions of the derivation processes will always proceed along the following pattern:

- firstly the description of the consequences of involved constructors occurring at the root side of the rule patterns (either just inside or just outside pattern occurrences),
- secondly the description of the consequences of involved constructors occurring at the leaf side of the rule patterns (either just inside or just outside pattern occurrences),
- finally the description of the consequences of involved constructors inside rule patterns.

In some of the descriptions the first and/or the second component is not present because there are no complications at the root and/or the leaf sides of the patterns.

The formal descriptions of how to transform rewrite systems will be given in the form of imperative programming style algorithms. The algorithms will be followed by an informal explanation of the actions in the algorithm. The algorithms are easy to understand for anyone who knows the programming language Pascal or something alike. The only detail that needs to be clarified is that rulearea, workarea1, and workarea2 are names of variables that can be bound to sets of rules. The rule set to be transformed is assumed to be bound to rulearea initially; the final rule set is also bound to this name.

Termination is easy to show for each of the algorithms that will be presented: each iteration construction is over a finite number of iterations which is fixed at the start.

### 6.4 Leaf Side Distinguishing

**Assumptions and Pre-conditions**

1. $f$ is the constructor to be distinguished in connection with operand position $p$ into $f_1$ and $f_2$ with $f_1$ and $f_2$ two new constructors which do not already occur in $C$,
2. \(\{C_{ch,1}, C_{ch,2}\}\) is a partition of \(C_{ch} = \rho(f, p)\), i.e.

- \(C_{ch} = C_{ch,1} \cup C_{ch,2}\),
- \(C_{ch,1} \cap C_{ch,2} = \emptyset\),

3. if \(\text{Root}(l) \in C_{ch,i}\) and \(\text{Root}(r) \in C_{ch,3-i}\) for some rule \(h = \langle l, r \rangle\) and some \(i \in \{1, 2\}\) then \(\text{Root}(l) \notin S\),

4. the rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with the same functionality, section 9.2.5).

**Explanation.** The assumptions imply that \(f\) is not a constant; constants do not have a leaf side context. If one of \(C_{ch,1}, C_{ch,2}\) equals the empty set then the distinguishing transformation reduces to the renaming of \(f\) (into \(f_1\) or \(f_2\)), which is a special case of this (and other) elementary transformations.

Pre-condition 3 is to ensure that eventual necessary specializations are always possible. Suppose \(\text{Root}(l) \in C_{ch,i}\) and \(\text{Root}(r) \in C_{ch,3-i}\) for some \(i\). Then if \(\text{Root}(l)\) occurs below \(f\) (at the right position \(p\)), \(f_i\) has to be replaced by \(f_{3-i}\) above \(\text{Root}(r)\) in the new domain with each application of the rule. This can only be done by extending the patterns of the rule with additional constructors above \(\text{Root}(l)\) and \(\text{Root}(r)\). These new rules will no longer match in terms in which \(\text{Root}(l)\) occurs as root of the term to be rewritten. Note that it is always possible to satisfy this condition by a root position distinguishing transformation (section 6.6).

Pre-condition 4, the restriction on collapsing rules, is to keep the derivation algorithm for rule sets simple. It is possible to design a derivation algorithm which is able to handle collapsing rules but this algorithm will be substantially more complicated. Moreover, the treatment of collapsing rules by this algorithm would often imply the conversion of collapsing rules into non-collapsing rules.

**Derivation of Vocabulary**

- \(C'' = C - \{f\} \cup \{f_1, f_2\}\);
- \(S'' = \text{if } f \in S \text{ then } S - \{f\} \cup \{f_1, f_2\} \text{ else } S\);
- \(\alpha'' = \alpha \triangleright \{f \leftarrow \bot, f_1 \leftarrow \alpha(f), f_2 \leftarrow \alpha(f)\}\);
- \(\rho' = \rho \triangleright \{(f, i) \leftarrow \bot,\)
  \[
  \langle f_k, p \rangle \leftarrow C_{ch,k},
  \]
  \[
  \langle f_k, q \rangle \leftarrow \rho(f, q) | \quad 1 \leq i \leq \alpha(f),
  \]
  \[
  k \in \{1, 2\},
  \]

6.4. LEAF SIDE DISTINGUISHING

\[ 1 \leq q \leq \alpha(f), q \neq p \];

- \( \rho'' = \rho' \uplus \{(c,m) \leftarrow \rho'(c,m) - \{f\} \cup \{f_1, f_2\} | c \in C, 1 \leq m \leq \alpha(c), f \in \rho'(c,m)\}; \)

- \( YV' = \bigcup_{yv \in YV} \text{if } yv \subseteq C_{ch} \land (yv \not\subseteq C_{ch,1} \lor yv \not\subseteq C_{ch,2}) \)
  \[ \text{then } \{yv \cap C_{ch,1}, yv \cap C_{ch,2}\} \]
  \[ \text{else } \{yv\}; \]

- \( YV'' = \{\text{if } f \in yv' \text{ then } yv' - \{f\} \cup \{f_1, f_2\} \text{ else } yv' \mid yv' \in YV'\}. \)

Definition of Domain Mapping

\[ \{\{f(v_1, \ldots, v_{p-1}, c_{1,i}(v_{p,1}, \ldots, v_{p,\alpha(c_{1,i})}), v_{p+1}, \ldots, v_{\alpha(f)}) \rightarrow f_1(v_1, \ldots, v_{p-1}, c_{1,i}(v_{p,1}, \ldots, v_{p,\alpha(c_{1,i})}), v_{p+1}, \ldots, v_{\alpha(f)}), \]

\[ f(v_1, \ldots, v_{p-1}, c_{2,i}(v_{p,1}, \ldots, v_{p,\alpha(c_{2,i})}), v_{p+1}, \ldots, v_{\alpha(f)}) \rightarrow f_2(v_1, \ldots, v_{p-1}, c_{2,i}(v_{p,1}, \ldots, v_{p,\alpha(c_{2,i})}), v_{p+1}, \ldots, v_{\alpha(f)}) | \]

\[ c_{1,i} \in C_{ch,1}, c_{2,i} \in C_{ch,2}, \]

\[ v_1, \ldots, v_{\alpha(f)} \text{ are distinct variables} \}

}\)\alpha.

Derivation of Rewrite System

(consequences for pattern's root side)
For each rule \( h = \langle l, r \rangle \) in rulearea Do:

If \( \text{Root}(l) \in C_{ch,1} \land \text{Root}(r) \in C_{ch,2} \lor \text{Root}(l) \in C_{ch,2} \land \text{Root}(r) \in C_{ch,1} \) Then:

- Replace \( h \) by the rule set which consists of the rules

\[ c_{1}(v_1, \ldots, v_{p_1-1}, l, v_{p_1+1}, \ldots, v_{\alpha(c_{1})}) \rightarrow c_{1}(v_1, \ldots, v_{p_1-1}, r, v_{p_1+1}, \ldots, v_{\alpha(c_{1})}), \]

for all \( \langle c_{1}, p_{1} \rangle \in \rho^{-1}(\text{Root}(l)) \) with \( \{v_1, \ldots, v_{p_1-1}, v_{p_1+1}, \ldots, v_{\alpha(c_{1})}\} \): distinct fresh variables, and each \( v_i \) such that it is maximally typed.

End If-Then.

End of For-iteration over rules;

(consequences for pattern's leaf side)
Set \text{workarea2} to \emptyset;

For each rule \( h = \langle l, r \rangle \) in rulearea Do:

Place \( h \) in an emptied \text{workarea1};

For each variable \( v \) in \text{Var}(l) Do:

...
If $(f,p) \in Par(v,l) \lor (f,p) \in Par(v,r)$ Then:
   If $\mu(v) \subseteq C_{ch,1}$ Then:
      Replace all occurrences of $f$ above $v$ (with $v$ at position $p$) by $f_1$ in all left hand sides and right hand sides of all rules in workarea1.
   End of If-Then;
   If $\mu(v) \subseteq C_{ch,2}$ Then:
      Replace all occurrences of $f$ above $v$ (with $v$ at position $p$) by $f_2$ in all left hand sides and right hand sides of all rules in workarea1.
   End of If-Then;
   If $\mu(v) \cap C_{ch,1} \neq \emptyset \land \mu(v) \cap C_{ch,2} \neq \emptyset$ Then:
      Replace each rule in workarea1 by two rules: one in which all occurrences of $f$ above $v$ (with $v$ at position $p$) have been replaced by $f_1$ and $v$ by $v_1$ with $\mu(v_1) = \mu(v) \cap C_{ch,1}$, and one in which all occurrences of $f$ above $v$ (with $v$ at position $p$) have been replaced by $f_2$ and $v$ by $v_2$ with $\mu(v_2) = \mu(v) \cap C_{ch,2}$.
   End of If-Then.
End of If-Then;
End of For-iteration over variables;
Add rules in workarea1 to workarea2.
End of For-iteration over rules;
Set rulearea to workarea2;

(consequences for pattern’s internal)
For each rule $h = (l,r)$ in rulearea Do:
   Replace all occurrences of $f$ in both $l$ and $r$ that have a subterm with root constructor $c \in V$ at position $p$ by constructor $f_1$ or $f_2$, depending on if $c \in C_{ch,1}$ or $c \in C_{ch,2}$;
   Replace all variables in both $l$ and $r$ the $\mu$-value of which contains $f$ with one with same $\mu$-value except that $f$ has been exchanged for $f_1$ and $f_2$.
End of For-iteration over rules.

Explanation. Let the assumptions be as stated in the beginning of this section. The consequences for the rules of a rewrite system for the original vocabulary can be explained informally as follows.

Consequences for pattern’s root side. If the root constructor of the left hand side pattern of a rule $h = (l,r)$ is in $C_{ch}$ then complications occur when left hand side root and right hand side root are not in the same subset of the partition, so if $\text{Root}(l) \in C_{ch,i} \land \text{Root}(r) \in C_{ch,3-i}$ for some $i \in \{1, 2\}$. In that case, the operation of the rule might cause a situation in which some occurrences of $f_1$ or $f_2$ have a constructor below them which is not in accordance with the partition of $C_{ch}$. The solution for this problem is an extension of the patterns of the rule with a constructor above the original root (the new constructor becomes root of the patterns) which changes $f_1$ into $f_2$ or vice versa. If $c = \text{Root}(l)$ is a constructor for which $\text{Root}(l) \in C_{ch,i}$ and $\text{Root}(r) \in C_{ch,3-i}$ for some $i$, then this rule has to be
replaced by \( n = |\rho^{-1}(c)| \) rules, one for each possible immediate root side context of \( c \). Obviously \( (f, p) \in \rho^{-1}(c) \). This \( f \) gives rise to a rule with a new \( f_i \) left hand side root and a new \( f_{3-i} \) right hand side root, for the particular \( i \in \{1, 2\} \). Eventual open operand positions of the new added roots have to be filled up with new variables which do not already occur in \( l \) and which are maximally typed. An example to illustrate this: suppose \( \text{Root}(l) \in C_{ch,1} \) and \( \text{Root}(r) \in C_{ch,2} \) and \( \rho^{-1}(\text{Root}(l)) = \{(f, 1), (g, 2), (h, 1)\} \), with \( f \) and \( g \) binary constructors and \( h \) an unary constructor. Then:

\[
l \rightarrow r
\]

has to be replaced by:

\[
\begin{align*}
f_1(l, v_1) & \rightarrow f_2(r, v_1), \\
g(v_2, l) & \rightarrow g(v_2, r), \\
h(l) & \rightarrow h(r).
\end{align*}
\]

The desired effect, in which the pair \( (f, p) \in \rho^{-1}(c) \) plays a special role, can be obtained most easily by first replacing the particular rule with a set of rules with new roots based upon \( \rho^{-1}(c) \) (in effect a root side specialization, section 9.2.5), and after that the replacement of occurrences of \( f \) by \( f_1 \) or \( f_2 \) (described below, with consequences for pattern’s internal)

It is not difficult to see that this measure is sufficient. After extending the patterns with an additional constructor it holds that \( \text{Root}(l) = \text{Root}(r) \), which implies that \( \text{Root}(l) \) and \( \text{Root}(r) \) are in the same subset of the partition of \( \rho(c, p) \). The distinguishing transformation will not change this: although \( f_1 \neq f_2 \), the partition is based on the constructor set before the transformation, so \( f_1 \) and \( f_2 \) are always in the same subset after the transformation.

Consequences for pattern’s leaf side. If left hand sides and/or right hand sides of the rules contain occurrences of the constructor \( f \) with a variable at position \( p \) then the appropriate action depends on the type (the \( \mu \)-value) of the variable and on the shape of the rule. In general the “multiplication” of a constructor will result in the multiplication of rules (in the examples \( f \) is assumed to be unary and \( p = 1 \)):

\[
g(f(v)) \rightarrow h(f(v))
\]

has to be replaced by two rules if \( \mu(v) \cap C_{ch,1} \neq \emptyset \) and \( \mu(v) \cap C_{ch,2} \neq \emptyset \):

\[
\begin{align*}
g(f_1(v_1)) & \rightarrow h(f_1(v_1)), \\
g(f_2(v_2)) & \rightarrow h(f_2(v_2)),
\end{align*}
\]

with \( \mu(v_1) = \mu(v) \cap C_{ch,1} \) and \( \mu(v_2) = \mu(v) \cap C_{ch,2} \). Also if \( v \) does not occur below \( f \) in the right hand side of the rule and \( \mu(v) \) overlaps both \( C_{ch,1} \) and \( C_{ch,2} \) multiplication of rules is necessary:
\[ g(f(v)) \rightarrow h(g(v)) \]

has to be replaced by the two rules:

\[ g(f_1(v_1)) \rightarrow h(g(v_1)), \]
\[ g(f_2(v_2)) \rightarrow h(g(v_2)), \]

and \( v_1 \) and \( v_2 \) as above. Likewise remarks for the situation in which \( v \) occurs not below \( f \) in the left hand side of the rule and \( \mu(v) \) overlaps both \( C_{ch,1} \) and \( C_{ch,2} \):

\[ g(h(v)) \rightarrow g(f(v)) \]

has to be replaced by the two rules:

\[ g(h(v_1)) \rightarrow g(f_1(v_1)), \]
\[ g(h(v_2)) \rightarrow g(f_2(v_2)), \]

and \( v_1 \) and \( v_2 \) as above. Multiple occurrences of \( f \) above different variables leads to combinations which should be taken care of:

\[ g(f(v_1), f(v_2)) \rightarrow h(f(v_1), f(v_2)) \]

has to be replaced by four rules if both \( \mu(v_1) \) and \( \mu(v_2) \) overlap both \( C_{ch,1} \) and \( C_{ch,2} \):

\[ g(f_1(v_{11}), f_1(v_{21})) \rightarrow h(f_1(v_{11}), f_1(v_{21})), \]
\[ g(f_1(v_{11}), f_2(v_{22})) \rightarrow h(f_1(v_{11}), f_2(v_{22})), \]
\[ g(f_2(v_{12}), f_1(v_{21})) \rightarrow h(f_2(v_{12}), f_1(v_{21})), \]
\[ g(f_2(v_{12}), f_2(v_{22})) \rightarrow h(f_2(v_{12}), f_2(v_{22})), \]

with:

\[ \mu(v_{11}) = \mu(v_1) \cap C_{ch,1}, \]
\[ \mu(v_{12}) = \mu(v_1) \cap C_{ch,2}, \]
\[ \mu(v_{21}) = \mu(v_2) \cap C_{ch,1}, \]
\[ \mu(v_{22}) = \mu(v_2) \cap C_{ch,2}. \]

Combinations have to do with different variables. If both variable occurrences below \( f \) are equal, i.e. a non-linear rule, then the number of combinations is reduced:

\[ g(f(v), f(v)) \rightarrow h(f(v), f(v)) \]
has to be replaced by the two rules:
\[ g(f_1(v_1), f_1(v_1)) \rightarrow h(f_1(v_1), f_1(v_1)), \]
\[ g(f_2(v_2), f_2(v_2)) \rightarrow h(f_2(v_2), f_2(v_2)), \]
and \( v_1 \) and \( v_2 \) as usual. Multiplication of rules is not necessary if the \( \mu \)-value of \( v \) is a subset of either \( C_{ch,1} \) or \( C_{ch,2} \). If \( \mu(v) \subseteq C_{ch,1} \) then:
\[ g(f(v)) \rightarrow h(f(v)) \]
can be replaced by:
\[ g(f_1(v)) \rightarrow h(f_1(v)). \]
Same reductions are possible in situations in which \( v \) does not occur below \( f \) in the left hand side or in the right hand side.

Consequences for pattern's internal. If left hand sides and/or right hand sides of the rules contain occurrences of the constructor \( f \) with no variable at position \( p \) then these occurrences have to be replaced by \( f_1 \) or \( f_2 \), depending on what constructor occurs at position \( p \) (one of \( C_{ch,1} \) or one of \( C_{ch,2} \)). If left hand sides and/or right hand sides of the rules contain variables the \( \mu \)-value of which contains \( f \) then these variables have to be replaced by variables with same \( \mu \)-value except that \( f \) has been exchanged for \( f_1 \) and \( f_2 \). Variables clearly belong to the leaf side bound of patterns but this action pertaining to variables will never lead to rule multiplication, that’s why this action is described at the “internal” clause of the algorithm.

The derivation algorithm may introduce variables which are not maximally typed (definition 5.20). This will be the case with variables that occur below \( f \) in the left hand side and not below \( f \) in the right hand side or vice versa. Non-maximally typed variables in left hand sides of rules are undesirable in certain circumstances because they impose an additional restriction on matches of the left hand side pattern. In section 9.2.4 is described how to eliminate this kind of variable occurrences. Note that variables \( v \) below the constructor \( f \) to be distinguished will not introduce non-maximally typed variables because of the reduction of the permissions of \( f \) at the position in question. Moreover: if \( \mu(v_i) \cap C_{ch,1} \subseteq \rho(f_1, p) \) (proper subset) then \( \mu(v) \subseteq \rho(f, p) \), so the non-maximality of \( v_i \) is not introduced in that case but simply propagated from \( v \).

6.5 Root Side Distinguishing

Assumptions and Pre-conditions

1. \( f \) is the constructor to be distinguished into \( f_1 \) and \( f_2 \) with \( f_1 \) and \( f_2 \) two new constructors which do not already occur in \( C \),

2. \( \{C_{pa,1}, C_{pa,2}\} \) is a partition of \( C_{pa} = \rho^{-1}(f) \), i.e.
• \( C_{pa} = C_{pa,1} \cup C_{pa,2} \),

• \( C_{pa,1} \cap C_{pa,2} = \emptyset \),

3. \( f \not\in S \),

4. the rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with the same functionality, section 9.2.5).

**Explanation.** If one of \( C_{pa,1}, C_{pa,2} \) equals the empty set then the distinguishing transformation reduces to the renaming of \( f \) (into \( f_1 \) or \( f_2 \)), which is a special case of this (and other) elementary transformations.

Pre-condition 3, the restriction \( f \not\in S \), is to support the division between root side distinguishing and root position distinguishing.

Pre-condition 4, the restriction on collapsing rules, is to keep the derivation algorithm for rule sets simple. It is possible to design a derivation algorithm which is able to handle collapsing rules but this algorithm will be substantially more complicated. Moreover, the treatment of collapsing rules by this algorithm would often imply the conversion of the collapsing rules into non-collapsing rules.

**Derivation of Vocabulary**

• \( C'' = C - \{f\} \cup \{f_1, f_2\} \);

• \( S'' = S \);

• \( \alpha'' = \alpha \uparrow \{f \leftarrow \bot, f_1 \leftarrow \alpha(f), f_2 \leftarrow \alpha(f)\} \);

• \( \rho' = \rho \uparrow \{\langle c, m \rangle \leftarrow \rho(c, m) - \{f\} \cup \{f_k\} | k \in \{1, 2\}, \langle c, m \rangle \in C_{pa,k}\} \);

• \( \rho'' = \rho' \uparrow \{\langle f, i \rangle \leftarrow \bot, \langle f_k, i \rangle \leftarrow \rho'(f, i) | 1 \leq i \leq \alpha(f), k \in \{1, 2\}\} \);

• \( YV'' = \bigcup_{yv \in YV} \text{if } f \in yv \)
  \[
  \text{then } \{y - \{f\}, yv - \{f\} \cup \{f_1\}, yv - \{f\} \cup \{f_2\}\}
  \]
  \[
  \text{else } \{yv\}.
  \]

**Definition of Domain Mapping**

\[
\begin{align*}
\{c_1,i(v_1, \ldots, v_{p_1-1}, f(v_{p_1,1}, \ldots, v_{p_1,\alpha(f)}), v_{p_1+1,1}, \ldots, v_{\alpha(c_1,i)}) \} & \rightarrow \\
\{c_1,i(v_1, \ldots, v_{p_1-1}, f_1(v_{p_1,1}, \ldots, v_{p_1,\alpha(f)}), v_{p_1+1,1}, \ldots, v_{\alpha(c_1,i)}) \}
\end{align*}
\]

\[
\begin{align*}
\{c_2,i(v_1, \ldots, v_{p_1-1}, f(v_{p_1,1}, \ldots, v_{p_1,\alpha(f)}), v_{p_1+1,1}, \ldots, v_{\alpha(c_2,i)}) \} & \rightarrow \\
\{c_2,i(v_1, \ldots, v_{p_1-1}, f_2(v_{p_1,1}, \ldots, v_{p_1,\alpha(f)}), v_{p_1+1,1}, \ldots, v_{\alpha(c_2,i)}) \}
\end{align*}
\]

\[
\langle c_1,i, p_i \rangle \in C_{ch,1}, \langle c_2,i, p_i \rangle \in C_{ch,2},
\]
6.5. ROOT SIDE DISTINGUISHING

\(v_1, \ldots, v_{\alpha(c_{1,1})}\) are distinct variables

}\end{array}

\]

Derivation of Rewrite System

(consequences for pattern's root side)

For each rule \(h = (l, r)\) in \(\text{rulearea}\) Do:

If \(\text{Root}(l) = f \land \text{Root}(r) = f\) Then:

Replace \(h\) by two rules which are derived from \(h\), one by substituting \(f_1\) for both roots, and one by substituting \(f_2\) for both roots.

End of If-Then;

If \(\text{Root}(l) = f \land \text{Root}(r) \neq f\) Then:

Replace \(h\) by two rules which are derived from \(h\), one by substituting \(f_1\) for the root of \(l\) and one by substituting \(f_2\) for the root of \(l\).

End of If-Then;

If \(\text{Root}(l) \neq f \land \text{Root}(r) = f\) Then:

If \(\rho^{-1}(\text{Root}(l)) \subseteq C_{pa,1}\) Then:

Replace \(h\) by a rule which is derived from \(h\) by substituting \(f_1\) for the root of \(r\).

End of If-Then;

If \(\rho^{-1}(\text{Root}(l)) \subseteq C_{pa,2}\) Then:

Replace \(h\) by a rule which is derived from \(h\) by substituting \(f_2\) for the root of \(r\).

End of If-Then;

If \(\rho^{-1}(\text{Root}(l)) \cap C_{pa,1} \neq \emptyset \land \rho^{-1}(\text{Root}(l)) \cap C_{pa,2} \neq \emptyset\) Then:

Let \(r_1\) be \(r\) in which the root node \(f\) has been replaced by \(f_1\) and let \(r_2\) be \(r\) in which the root node \(f\) has been replaced by \(f_2\). Replace \(h\) by the two rule sets which consists of the rules:

\[c_t(v_1, \ldots, v_{p_1-1}, l, v_{p_1+1}, \ldots, v_{\alpha(c_t)}) \rightarrow c_t(v_1, \ldots, v_{p_1-1}, r_1, v_{p_1+1}, \ldots, v_{\alpha(c_t)})\]

for all \(\langle c_t, p_t \rangle \in C_{pa,1}\) with \(\{v_1, \ldots, v_{p_1-1}, v_{p_1+1}, \ldots, v_{\alpha(c_t)}\}\): distinct fresh variables, and each \(v_i\) such that it is maximally typed,

\[c_t(v_1, \ldots, v_{p_1-1}, l, v_{p_1+1}, \ldots, v_{\alpha(c_t)}) \rightarrow c_t(v_1, \ldots, v_{p_1-1}, r_2, v_{p_1+1}, \ldots, v_{\alpha(c_t)})\]

for all \(\langle c_t, p_t \rangle \in C_{pa,2}\) with \(\{v_1, \ldots, v_{p_1-1}, v_{p_1+1}, \ldots, v_{\alpha(c_t)}\}\): distinct fresh variables, and each \(v_i\) such that it is maximally typed.

End of If-Then.

End of For- iteration over rules;

(consequences for pattern’s leaf side)

Set \(\text{workarea}_2\) to \(\emptyset\);

For each rule \(h = (l, r)\) in \(\text{rulearea}\) Do:

Place \(h\) in an emptied \(\text{workarea}_1\);

For each variable \(v\) in \(\text{Var}(l)\) Do:

Let \(P\) be the set \(\text{Par}(v, l) \cup \text{Par}(v, r)\);
If \( P \subseteq C_{pa,1} \) Then:
Replace all occurrences of \( v \) in all rules in \( \text{workarea1} \) with a new variable
with same \( \mu \)-value, except that \( f \) has been exchanged for \( f_1 \);
End of If-Then;
If \( P \subseteq C_{pa,2} \) Then:
Replace all occurrences of \( v \) in all rules in \( \text{workarea1} \) with a new variable
with same \( \mu \)-value, except that \( f \) has been exchanged for \( f_1 \);
End of If-Then;
If \( P \cap C_{pa,1} \neq \emptyset \land P \cap C_{pa,2} \neq \emptyset \) Then:
Replace all occurrences of \( v \) in all rules in \( \text{workarea1} \) with a new variable
with same \( \mu \)-value, except that \( f \) has been removed from this \( \mu \)-value;
For each rule \( h \) in \( \text{workarea1} \)
Add a rule to \( \text{workarea2} \) in which all occurrences of \( v \) are replaced
by either \( f_1(v_1, \ldots, v_{\alpha(f)}) \) or \( f_2(v_1, \ldots, v_{\alpha(f)}) \), depending on if the
particular parent-index pair is in \( C_{pa,1} \) or \( C_{pa,2} \).
End For-iteration over rules in \( \text{workarea2} \)
End If-Then
End of For-iteration over variables;
Add rules in \( \text{workarea1} \) to \( \text{workarea2} \).
End of For-iteration over rules;
Set \( \text{rulearea} \) to \( \text{workarea2} \).

(\text{consequences for pattern's internal})
For each rule \( h = \langle l, r \rangle \) in \( \text{rulearea} \) Do:
Replace all occurrences of \( f \) in both \( l \) and \( r \) at non-root positions by constructor
\( f_1 \) or \( f_2 \), depending on if the direct parent-position pair is in \( C_{pa,1} \) or \( C_{pa,2} \);
End of For-iteration over rules.

\text{Explanation}. Let the assumptions be as stated in the beginning of this section.
The consequences for the rules of a rewrite system for the original vocabulary can
be explained informally as follows.

\text{Consequences for pattern's root side}. If left hand sides or right hand sides of the
rules contain occurrences of the constructor \( f \) at root positions then the appropriate
action depends on the nature of both root-constructors of the patterns of the rule.
If the left hand side root equals \( f \) then the rule simply has to be multiplied, one
new rule for \( f_1 \) and one new rule for \( f_2 \). An eventual right hand side \( f \)-root simply
follows the left hand side:

\[ f(...) \rightarrow c(...) \]

has to be replaced by:

\[ f_1(...) \rightarrow c(...), \]

\[ f_2(...) \rightarrow c(...). \]
and:

\[ f(...) \rightarrow f(...) \]

has to be replaced by:

\[ f_1(...) \rightarrow f_1(...), \]
\[ f_2(...) \rightarrow f_2(...). \]

A right hand side root which equals \( f \) and a left hand side which doesn't, needs special attention. The constructors \( f \) and \( c \) in:

\[ c(...) \rightarrow f(...) \]

are, according to theorem 5.28, related as follows: \( f \) is allowed to occur at least at all positions where \( c \) is allowed to occur. The appropriate action depends on the relation between the allowed positions of \( c, f_1, f_2 \) in the new vocabulary. If the positions where \( c \) is allowed to occur is a subset of the positions where \( f_1 \) is allowed to occur then the root of the right hand side, has to be replaced by \( f_1 \). If the positions where \( c \) is allowed to occur is a subset of the positions were \( f_2 \) is allowed to occur then the root of the right hand side, has to be replaced by \( f_2 \). If both sets of positions where \( f_1 \) and \( f_2 \) are allowed to occur have a non-empty intersection with the positions where \( c \) is allowed to occur then the rule has to be multiplied and an additional root-constructor has to be added (above the root) to restrict the context in which the rule is allowed to match. A rationale for this action is the following. Replacing the single rule:

\[ c(...) \rightarrow f(...) \]

with the two rules:

\[ c(...) \rightarrow f_1(...), \]
\[ c(...) \rightarrow f_2(...), \]

will introduce unwanted functionality. Occurrences of the pattern \( c(...) \) in terms to be rewritten can be replaced by both \( f_1(...) \) and \( f_2(...) \), while only gone is appropriate in each situation. To chose between the two the rules have to be extended with a new root constructor to restrict the contexts of both rules in which they are allowed to match.

*Consequences for pattern's leaf side.* If left hand sides and/or right hand sides of rules contain occurrences of variables the \( \mu \)-value of which contains \( f \) then the appropriate action depends on the nature of the constructors which appear directly above these variables. If all constructor-position pairs directly above a certain variable with a permission for \( f \) occur in one of \( C_{pa,1} \) or \( C_{pa,2} \), then these variables have
to be replaced by variables with same \( \mu \)-value, except that \( f \) has been exchanged for \( f_1 \) or \( f_2 \), depending on if it is \( C_{pa,1} \) or \( C_{pa,2} \). If the constructor-position pairs directly above a certain variable with a permission of \( f \) occur in both \( C_{pa,1} \) and \( C_{pa,2} \) then the rule has to be multiplied. The original rule has to be maintained with a variable from which the permission for \( f \) has been precluded. Moreover, a new rule has to be added which takes care of the difference in the root constructors of the terms that can be bound against the variable in question. An example to illustrate this. Suppose \( f \in \mu(v) \) and \( \langle g, 1 \rangle \) and \( \langle h, 1 \rangle \) do not occur in the same \( C_{pa,i} \). Then:

\[
g(v) \rightarrow h(v)
\]

has to be replaced by:

\[
g(v') \rightarrow h(v'),
\]

\[
g(f_1(v_1, ..., v_{\alpha(f)})) \rightarrow h(f_2(v_1, ..., v_{\alpha(f)})),
\]

with \( \mu(v') = \mu(v) - \{f\} \) and each \( v_1, ..., v_{\alpha(f)} \) such that it is maximally typed. The actions for the pattern's leaf side will always introduce non-maximally typed variables: the expression \( \mu(v') = \mu(v) - \{f\} \) is easily seen to lead to a non-maximally typed variable occurrence.

Consequences for pattern's internal. If left hand sides or right hand sides of the rules contain occurrences of the constructor \( f \) at non-root positions then these occurrences have to be replaced by \( f_1 \) or \( f_2 \), depending on what constructor (and position) occurs immediately above it (a pair from \( C_{pa,1} \) or one from \( C_{pa,2} \)).

6.6 Root Position Distinguishing

Assumptions and Pre-conditions

1. \( F \) is the set of constructors to be distinguished into constructors from \( F_1 \) and \( F_2 \). \( F_1 \) and \( F_2 \) are disjunct sets of new constructors which do not already occur in \( C \), \( F_1 \) contains the new names for root occurrences of elements of \( F \) and \( F_2 \) contains the new names for non-root occurrences of elements of \( F \). There is a fixed correspondence between constructors from \( F \) on the one hand and their replacements from \( F_1 \) and \( F_2 \) on the other hand and \( |F| = |F_1| = |F_2| \). The correspondence between constructors from \( F \) and replacements of these constructors in \( F_1 \) and \( F_2 \) is fixed in the relation \( R \subseteq F \times (F_1 \cup F_2) \). If \( f_1 \) and \( f_2 \) are the replacements of \( f \) then \( \langle f, f_1 \rangle \in R \) and \( \langle f, f_2 \rangle \in R \), notation: \( fRf_1 \) and \( fRf_2 \),

2. if there are rules with a right hand side root in \( F \) then the root of the left hand side is either in \( F \) or specific (section 6.3.2),
3. the rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with the same functionality, section 9.2.5).

Explanation. The root position distinguishing is defined for sets of constructors to be distinguished, as opposed to root side distinguishing, which was only defined for one constructor to be distinguished. The reason for this is that root position distinguishing is sometimes only possible if more than one constructor is distinguished simultaneously. The situations in which a set of constructors has to be distinguished simultaneously is determined by pre-condition 3. This condition depends, unlike most other pre-conditions, on the shape of the rules to be transformed.

If a constructor \( f \in F \) is specific (section 6.3.2) then the root position transformation reduces to the renaming of \( f \) into \( f_1 \) or \( f_2 \) (depending on if \( f \) is root-specific or non-root-specific, section 6.3.2).

Pre-condition 2, the condition with regard to root nodes of rules and the required being in \( F \) or being specific, is to guarantee that a derivation of a simulating rule set is possible. If the right hand side root of a rule is in \( F \) then the root node of the left hand side pattern of the rule has to "decide" whether the right hand side root has to be replaced by \( f_1 \) or \( f_2 \) (or both). This decision can only be made if the left hand side root is already specific or if it will be made specific (by this transformation); it has to be in \( F \) therefore.

Pre-condition 3, the restriction on collapsing rules, is to keep the derivation algorithm for rule sets simple. It is possible to design a derivation algorithm which is able to handle collapsing rules but this algorithm will be substantially more complicated. Moreover, the treatment of collapsing rules by this algorithm would often imply the conversion of the collapsing rules into non-collapsing rules.

Derivation of Vocabulary

- \( C'' = C - F \cup F_1 \cup F_2 \);
- \( S'' = S - F \cup F_1 \);
- \( \alpha'' = \alpha \dagger \{ f \leftarrow \bot, \)
  \[ f_1 \leftarrow \alpha(f), \]
  \[ f_2 \leftarrow \alpha(f) | \]
  \[ f \in F, f_1 \in F_1, f_2 \in F_2, \]
  \[ fRf_1, fRf_2 \}; \]
- \( \rho' = \rho \dagger \{(c, m) \leftarrow \rho(c, m) - \{ f \} \cup \{ f_2 \} | \)
  \[ c \in C, 1 \leq m \leq \alpha(c), \]
  \[ f \in F \cup \rho(c, m), f_2 \in F_2, fRf_2 \}; \]
\[ \rho'' = \rho' \upharpoonright \{ \langle f, i \rangle \leftarrow \bot, \]
\[ \langle f_k, i \rangle \leftarrow \rho'(f, i) \mid f \in F, 1 \leq i \leq \alpha(f), \]
\[ k \in \{1, 2\}, \]
\[ f_k \in F_k, fRF_k \}; \]
\[ \{ i f \in yv \ \text{then} \ \forall v \in \left\{ f \cup \{ f_2 \} \right\} \left\{ \forall v \in YV \right\}. \]

**Definition of Domain Mapping**

The description of the domain mapping associated with this transformation cannot be done with term rewriting systems for reasons mentioned earlier (section 6.3.4). A description with induction on the structure of terms will be given of \( \phi : TG_{\Sigma} \rightarrow TG_{\Sigma'} \), using auxiliary function \( \phi' : RG_{\Sigma} \rightarrow RG_{\Sigma'} \) with:

- \( \phi(f(t_1, \ldots, t_{\alpha(f)})) = f_1(\phi'(t_1), \ldots, \phi'(t_{\alpha(f)})) \)
  for all \( f \in F, f_1 \in F_1 \) with \( fRF_1 \),
- \( \phi(c(t_1, \ldots, t_{\alpha(c)})) = c(\phi'(t_1), \ldots, \phi'(t_{\alpha(c)})) \)
  for all \( c \not\in F \),
- \( \phi'(f(t_1, \ldots, t_{\alpha(f)})) = f_2(\phi'(t_1), \ldots, \phi'(t_{\alpha(f)})) \)
  for all \( f \in F, f_2 \in F_2 \) with \( fRF_2 \),
- \( \phi'(c(t_1, \ldots, t_{\alpha(c)})) = c(\phi'(t_1), \ldots, \phi'(t_{\alpha(c)})) \)
  for all \( c \not\in F \).

**Explanation.** The system is designed such that \( \phi \) is applied only to top-level terms and \( \phi' \) is applied only to sub-terms. Root occurrences of \( f \) are replaced by \( f_1 \) and internal occurrences of \( f \) by \( f_2 \).

**Derivation of Rewrite System**

(consequences for pattern's root side)

For each rule \( h = (l, r) \) in rulearea Do:

- If \( \text{Root}(l) = f \in F \land \text{Root}(r) = g \in F \) Then:
  - Replace \( h \) by two rules which are derived from \( h \), one by substituting \( f_1 \) for \( f \) and \( g_1 \) for \( g \), and one by substituting \( f_2 \) for \( f \) and \( g_2 \) for \( g \) with \( fRF_1, fRF_2, gRG_1, gRG_2 \).

End of If-Then;

- If \( \text{Root}(l) = f \in F \land \text{Root}(r) = g \not\in F \) Then:
Replace \( h \) by two rules which are derived from \( h \), one by substituting \( f_1 \) for the root of \( l \) and one by substituting \( f_2 \) for the root of \( l \), with \( fRf_1, fRf_2 \).

End of If-Then;
If \( \text{Root}(l) = g \notin F \land \text{Root}(r) = f \in F \) Then:
   If \( g \) is root-specific Then
      Replace \( h \) by a rule which is derived from \( h \) by substituting \( f_1 \) for the root of \( r \) with \( fRf_1 \).
   End of If-Then;
   If \( g \) is non-root-specific Then
      Replace \( h \) by a rule which is derived from \( h \) by substituting \( f_2 \) for the root of \( r \) with \( fRf_2 \).
   End of If-Then.
End of If-Then.
End of For-iteration over rules;

(consequences for pattern’s internal)
For each rule \( h = (l, r) \) in rulearea Do:
   Replace all occurrences of each \( f \in F \) in both \( l \) and \( r \) at non-root positions by constructor \( f_2 \in F_2 \) with \( fRf_2 \);
   Replace all variables the \( \mu \)-value of which contains an \( f \in F \) with variables with same \( \mu \)-value except that \( f \in F^\prime \) has been exchanged for \( f_2 \in F_2 \) with \( fRf_2 \).
End of For-iteration over rules.

Explanation. Let the assumptions be as stated in the beginning of this section. The consequences for the rules of a rewrite system for the original vocabulary are rather simple for this transformation.

Consequences for pattern’s root side. If left hand sides or right hand sides of the rules contain occurrences of the constructor \( f \in F \) at root positions then there are complications and the appropriate action depends on the nature of both root-constructors of the patterns of the rule. If both left and right hand side roots are non-specific constructors (section 6.3.2) which have to be distinguished then the rule has to be multiplied: one version with root-specific roots and one version with non-root-specific roots, because the original pattern may match at root positions and at non-root positions:

\[ f(\ldots) \rightarrow g(\ldots) \]

has to be replaced by:

\[ f_1(\ldots) \rightarrow g_1(\ldots), \]
\[ f_2(\ldots) \rightarrow g_2(\ldots), \]

with \( fRf_1, fRf_2, gRg_1, gRg_2 \). Note that \( f \in F \) and \( g \in F \) need not necessarily be different constructors. If the right hand side root is not member of \( F \) the procedure is similar: the rule has to be multiplied because the left hand side pattern can match at root-positions and at non-root positions:
\[ f(\ldots) \rightarrow g(\ldots) \]

has to be replaced by:

\[ f_1(\ldots) \rightarrow g(\ldots), \]
\[ f_2(\ldots) \rightarrow g(\ldots), \]

with \( fRf_1, fRf_2 \). If the right hand side root is member of \( F \) (i.e. has to be distinguished) and the left hand side root not then the left hand side needs to be specific since the pattern must be restricted to match either at root positions or at non-root positions in order to be able to substitute the right replacement for the right hand side root.

*Consequences for pattern's leaf side.* No special actions are necessary in this case. The reason is that the constructors \( f \in F \) to be distinguished will always be replaced by \( f_2 \in F_2 \), when it occurs as root node of a subterm which matches against the variable of a pattern, because patterns consisting of one single variable are prohibited by the general requirements for rules and by the non-collapsing requirement for this transformation. It is therefore impossible that similar inconsistencies are introduced as with root side distinguishing.

*Consequences for pattern's internal.* If left hand sides or right hand sides of the rules contain occurrences of the constructor \( f \in F \) at non-root positions then these occurrences have to be replaced by the corresponding \( f_2 \in F_2 \), since the patterns will never be matched against a term in which an \( f_1 \in F_1 \) occurs at an internal position. If left hand sides or right hand sides of rules contain occurrences of variables the \( \mu \)-value of which contains \( f \in F \) then these variables have to be replaced by variables with same \( \mu \)-value, except that \( f \) has been exchanged for \( f_2 \). Variables will not occur on the root positions of either side of rules because of the definition of rules (definition 5.21) in conjunction with the restriction on collapsing rules (third precondition). Variables will therefore never be matched against a subterm with \( f_1 \) as root constructor.

### 6.7 Leaf Side Identification

**Assumptions and Pre-conditions**

1. \( f_1 \) and \( f_2 \) are the constructors to be identified in connection with operand position \( p \) into \( f \) with \( f \) a new constructor which does not already occur in \( C \).
2. \( \rho(f_1, p) = C_{ch,1}, \rho(f_2, p) = C_{ch,2}; \)
3. \( C_{ch} = C_{ch,1} \cup C_{ch,2}; \)
4. \( \alpha(f_1) = \alpha(f_2); \)
5. \( \rho(f_1, p) \cap \rho(f_2, p) = \emptyset; \)
6. \( \forall 1 \leq i \leq \alpha(f_1), i \neq p \ [\rho(f_1, i) = \rho(f_2, i)] \),
7. \( \rho^{-1}(f_1) = \rho^{-1}(f_2) \),
8. \( f_1 \in S \Leftrightarrow f_2 \in S \).

*Explanation.* Pre-condition 4 is a necessary and self-evident requirement for two constructors to be identifiable. The Pre-conditions 5 and 6 guarantee that it is possible to reconstruct the origin of an identified constructor out of the identity of the surrounding constructors.

**Derivation of Vocabulary**

- \( C'' = C - \{f_1, f_2\} \cap \{f\} \);
- \( S'' = \text{if } f_1 \in S \lor f_2 \in S \text{ then } S - \{f_1, f_2\} \cup \{f\} \text{ else } S \);
- \( \alpha'' = \alpha \uparrow \{f_1 \leftarrow \bot, f_2 \leftarrow \bot, f \leftarrow \alpha(f_1)\} \);
- \( \rho' = \rho \uparrow \{(f, i) \leftarrow \bot, \rho(f_1, i) \cup \rho(f_2, i) | k \in \{1, 2\}, 1 \leq i \leq \alpha(f_1)\} \);
- \( \rho'' = \rho \uparrow \{(c, m) \leftarrow \rho'(c, m) - \{f_1, f_2\} \cup \{f\} | c \in C, 1 \leq m \leq \alpha(c), \rho'(c, m) \cap \{f_1, f_2\} \neq \emptyset\} \);
- \( YV'' = \{ \text{if } f_1 \in yv \lor f_2 \in yv \text{ then } yv - \{f_1, f_2\} \cup \{f\} \text{ else } yv | yv \in YV\} \).

**Definition of Domain Mapping**

\[
\begin{align*}
[f_1(v_1, ..., v_{p-1}, c_{1,i}(v_{p,1}, ..., v_{p,\alpha(c_{1,i})}), v_{p+1}, ..., v_{\alpha(f_1)})]_o \rightarrow \\
f(v_1, ..., v_{p-1}, c_{1,i}(v_{p,1}, ..., v_{p,\alpha(c_{1,i})}), v_{p+1}, ..., v_{\alpha(f_1)}), \\
f_2(v_1, ..., v_{p-1}, c_{2,i}(v_{p,1}, ..., v_{p,\alpha(c_{2,i})}), v_{p+1}, ..., v_{\alpha(f_2)}) \rightarrow \\
f(v_1, ..., v_{p-1}, c_{2,i}(v_{p,1}, ..., v_{p,\alpha(c_{2,i})}), v_{p+1}, ..., v_{\alpha(f_2)}), \\
c_{1,i} \in C_{ch,1}, c_{2,i} \in C_{ch,2}, \\
v_1, ..., v_{\alpha(f)} \text{ are distinct variables}
\end{align*}
\]
Derivation of Rewrite System

(consequences for pattern’s internal)
For each rule $h = (l, r)$ in rulearea Do:

Replace all occurrences of $f_1$ and $f_2$ in both $l$ and $r$ by an occurrence of the constructor $f$;

Replace all variables in both $l$ and $r$ the $\mu$-value of which contains $f_1$ and/or $f_2$, with one with same $\mu$-value except that $f_1$ and/or $f_2$ has been exchanged for $f$.

End of For-iteration over rules.

explanation. Let the assumptions be as stated in the beginning of this section. The consequences for the rules of a rewrite system for the original vocabulary are surprisingly simple for this transformation, thanks to the availability of typed variables, and more specifically: variables which can have a narrower type than the type of the position they occupy. The transformation may therefore introduce variable occurrences which are non-maximally typed (definition 5.20). In section 9.2.4 is described how to eliminate variable occurrences of this kind. The strategy for the identification in connection with leaf context is simply to replace all occurrences of the constructors $f_1$ and $f_2$ with the constructor $f$. The only situation in which complications may be expected to occur is the case of constructors $f_1$ or $f_2$ with a variable on the identification position $p$. For instance:

$$f_1(v) \rightarrow g(v).$$

Transforming this rule according the given strategy means replacing $f_1$ by the more general $f$, which is the identification of $f_1$ and $f_2$. If there was no typing of variables the transformed rule:

$$f(v) \rightarrow g(v)$$

would imply the functionality of the rule:

$$f_2(v) \rightarrow g(v),$$

which is not correct. However, typing of $v$ will restrict the possible matches against $v$ to the functionality of the original rule:

$$f_1(v) \rightarrow g(v).$$

Note that the left hand side of this transformed rule contains a variable which is non-maximally typed. No special actions are necessary for the pattern’s root side and the pattern’s leaf side. Complications at the pattern’s root side are simply non-existent in the case of leaf side identification; problems similar to the problems which occur with leaf side distinguishing do not occur in this case because identifications cannot cause inconsistencies based on root constructors of patterns. Complications at the pattern’s leaf side(s) are resolved by the use on non-maximally typed variables. This means actually that these complications are only postponed. If a subsequent transformation requires maximally typed variables, the complications will occur in the necessary application of schemes to remove non-maximally typed variables (section 9.2.4).
6.8 Root Side Identification

Assumptions and Pre-conditions

1. \( f_1 \) and \( f_2 \) are the constructors to be identified into \( f \) with \( f \) a new constructor which does not already occur in \( C \),

2. \( \rho^{-1}(f_1) = C_{pa,1}, \rho^{-1}(f_2) = C_{pa,2}, \)

3. \( C_{pa} = C_{pa,1} \cup C_{pa,2}, \)

4. \( \alpha(f_1) = \alpha(f_2), \)

5. \( \forall 1 \leq i \leq \alpha(f_1) [\rho(f_1, i) = \rho(f_2, i)], \)

6. \( \rho^{-1}(f_1) \cap \rho^{-1}(f_2) = \emptyset, \)

7. \( f_1 \not\in S, f_2 \not\in S. \)

Explanation. Pre-condition 4 is a necessary and self-evident requirement for two constructors to be identifiable. The Pre-conditions 5 and 6 guarantee that it is possible to reconstruct the origin of an identified constructor based on the identity of the surrounding constructors. Pre-condition 7 is, like pre-condition 4, of root side distinguishing (section 6.5), to support the division between root side identification and root position identification.

Derivation of Vocabulary

- \( C'' = C - \{f_1, f_2\} \cap \{f\}; \)

- \( S'' = S; \)

- \( \alpha'' = \alpha \uparrow \{f_1 \leftarrow \bot, f_2 \leftarrow \bot, f \leftarrow \alpha(f_1)\}; \)

- \( \rho' = \rho \uparrow \{(f_k, i) \leftarrow \bot, \)

\[ (f, i) \leftarrow \rho(f_1, i) \cup \rho(f_2, i) \mid \]

\[ k \in \{1, 2\}, \]

\[ 1 \leq i \leq \alpha(f_1)\}; \)

- \( \rho'' = \rho' \uparrow \{(c, m) \leftarrow \rho'(c, m) - \{f_k\} \cup \{f\} \mid \)

\[ c \in C, 1 \leq m \leq \alpha(c), \]

\[ k \in \{1, 2\}, \]

\[ f_k \in \rho'(c, m)\}; \)
- $YV'W = \{\text{if } f_1 \in yv \lor f_2 \in yv \text{ then } yv - \{f_1, f_2\} \cup \{f\} \text{ else } yv \mid yv \in YV\}$.

**Definition of Domain Mapping**

\[\{(c_{1,i}(v_1, \ldots, v_{p_i-1}, f_1(v_{p_i,1}, \ldots, v_{p_i,\alpha(f_1)}), v_{p_i+1}, \ldots, v_{\alpha(c_{1,i})})),\]
\[c_{1,i}(v_1, \ldots, v_{p_i-1}, f(v_{p_i,1}, \ldots, v_{p_i,\alpha(f)}), v_{p_i+1}, \ldots, v_{\alpha(c_{1,i})})\}
\[c_{2,i}(v_1, \ldots, v_{p_i-1}, f_2(v_{p_i,1}, \ldots, v_{p_i,\alpha(f_2)}), v_{p_i+1}, \ldots, v_{\alpha(c_{2,i})})\}
\[c_{2,i}(v_1, \ldots, v_{p_i-1}, f(v_{p_i,1}, \ldots, v_{p_i,\alpha(f)}), v_{p_i+1}, \ldots, v_{\alpha(c_{2,i})})\}
\[(c_{1,i}, p_i) \in C_{ch,1}, (c_{2,i}, p_i) \in C_{ch,2},
\]
\[v_1, \ldots, v_{\alpha(c_{i,i})} \text{ are distinct variables}\}
\]

\[\} \alpha.\]

**Derivation of Rewrite System**

*(consequences for pattern’s root side)*

For each rule $h = (l, r)$ in rulearea Do:

If $Root(l) = f_1 \lor Root(r) = f_1$ Then:

Replace $f_1$ with $f$ in $l$ and/or $r$, name result $h' = (l', r')$;

If $Root(l') = f$ Then:

Replace the modified rule $h'$ with the $n = |C_{pa,1}|$ rules:

\[\langle c_i(v_1, \ldots, v_{p_i-1}, l', v_{p_i+1}, \ldots, v_{\alpha(c_i)})\rangle, c_i(v_1, \ldots, v_{p_i-1}, r', v_{p_i+1}, \ldots, v_{\alpha(c_i)})\rangle,\]

one for each $(c_i, p_i) \in C_{pa,1}$, with $\{v_1, \ldots, v_{p_i-1}, v_{p_i+1}, \ldots, v_{\alpha(c_i)}\}$: distinct fresh variables and each $v_i$ such that it is maximally typed.

End of If-Then.

End of If-Then;

If $Root(l) = f_2 \lor Root(r) = f_2$ Then:

Replace $f_2$ with $f$ in $l$ and/or $r$, name result $h' = (l', r')$;

If $Root(l') = f$ Then:

Replace the modified rule $h'$ with the $n = |C_{pa,2}|$ rules:

\[\langle c_i(v_1, \ldots, v_{p_i-1}, l', v_{p_i+1}, \ldots, v_{\alpha(c_i)})\rangle, c_i(v_1, \ldots, v_{p_i-1}, r', v_{p_i+1}, \ldots, v_{\alpha(c_i)})\rangle,\]

one for each $(c_i, p_i) \in C_{pa,2}$, with $\{v_1, \ldots, v_{p_i-1}, v_{p_i+1}, \ldots, v_{\alpha(c_i)}\}$: distinct fresh variables and each $v_i$ such that it is maximally typed.

End of If-Then.

End of If-Then.

End of For-iteration over rules;

*(consequences for pattern’s internal)*

For each rule $h = (l, r)$ in rulearea Do:

Replace all occurrences of $f_1$ and $f_2$ in both $l$ and $r$ at non-root positions by $f$;

Replace all variables in both $l$ and $r$ the $\mu$-value of which contains $f_1$ or $f_2$ or both with one with same $\mu$-value except that $f_1$ and/or $f_2$ has been exchanged for $f$. 
6.8. ROOT SIDE IDENTIFICATION

End of For-iteration over rules.

Explanation. Let the assumptions be as stated in the beginning of this section. The consequences for the rules of a rewrite system for the original vocabulary can be explained informally as follows.

Consequences for pattern's root side. If left hand sides or right hand sides of the rules contain occurrences of the constructor \( f_1 \) or \( f_2 \) at root-positions then there are complications. Note that an \( f_1 \) as left hand side root and an \( f_2 \) as right hand side root, or vice versa, is ruled out by the condition of theorem 5.28 in conjunction with the sixth pre-condition of this transformation. The proper action is: first replacing the one or two \( f_1 \)-roots by \( f \) or replacing the one or two \( f_2 \)-roots by \( f \). After that replacing the new rule with \( n \) rules, one for each possible context for the \( f_1 \) or \( f_2 \) if the root of the left hand side of the rule is equal to \( f \). If the root of the left hand side of the rule is not equal to \( f \) no additional context restrictions are necessary for reasons that will be explained below. The \( n \) rules can be defined as follows: for each \( \langle c, i \rangle \in \rho^{-1}(f_k) \) extend both patterns of the original rule above the root with \( c \) and the pattern at position \( i \) of \( c \). Remaining positions of \( c \) have to be filled up with fresh variables, corresponding variables at left-right corresponding positions. A rationale for this action is the following: when replacing \( f_1 \) by \( f \) in rules of the form:

\[
f_1(... \rightarrow f_1(...) \]

and:

\[
f_1(... \rightarrow c(... \]

there may be introduced an unwanted functionality which can be described by:

\[
f_2(... \rightarrow f_2(...) \]

and:

\[
f_2(... \rightarrow c(... \]

To restrict the functionality of the transformed rule to the situations which were originally covered by \( f_1 \), a constructor has to be added above the root that restricts the context in which the rule is allowed to operate. Rules of the form:

\[
c(... \rightarrow f_1(...) \]

will not introduce unwanted functionality when \( f_1 \) is replaced by \( f \) because the positions where \( c \) is allowed to occur is a subset of the positions where \( f_1 \) is allowed to occur (theorem 5.28). In other words: the meaning of \( f \) in:

\[
c(... \rightarrow f(...) \]

is completely determined by the context of \( c \) and \( f \).

Consequences for pattern’s leaf side. No special actions are necessary for the pattern’s leaf side. Complications at the pattern’s leaf side are simply non-existent in the case of root side identification; problems similar to the problems which occur with root side distinguishing do not occur in this case because identifications cannot cause inconsistencies based on constructors just above variables.

Consequences for pattern’s internal. If left hand sides and/or right hand sides of the rules contain occurrences of the constructor \( f_1 \) or \( f_2 \) at non-root positions then these occurrences can safely be replaced by constructor \( f \). If left hand sides and/or right hand sides of the rules contain variables the \( \mu \)-value of which contains \( f_1 \) or \( f_2 \) then these variables have to be replaced by variables with same \( \mu \)-value except that \( f_1 \) or \( f_2 \) has been exchanged for \( f \).

6.9 Root Position Identification

Assumptions and Pre-conditions

1. \( F_1 \) and \( F_2 \) are the sets of constructors to be identified into constructors from \( F \) with \( F \) a set of new constructors which do not already occur in \( C \) (\( F_1 \) and \( F_2 \) have to be disjunct of course). There is a fixed correspondence between constructors from \( F_1 \) and \( F_2 \) on the one hand and their replacements on the other hand and \( |F_1| = |F_2| = |F| \). The correspondence between constructors from \( F_1 \) and \( F_2 \) and replacements of these constructors in \( F \) is fixed in the relation \( R \subseteq (F_1 \cup F_2) \times F \). If \( f \) is the replacement of \( f_1 \) and \( f_2 \) then \( (f_1, f) \in R \) and \( (f_2, f) \in R \), notation \( f_1 R f \) and \( f_2 R f \),

2. \( \forall f_1 \in F_1, f_2 \in F_2, f \in F [f_1 R f \land f_2 R f \Rightarrow \alpha(f_1) = \alpha(f_2)] \),

3. \( \forall f_1 \in F_1, f_2 \in F_2, f \in F [f_1 R f \land f_2 R f \Rightarrow \forall 1 \leq i \leq \alpha(f_1) [\rho(f_1, i) = \rho(f_2, i)]] \),

4. each \( f_1 \in F_1 \) is root-specific,

5. each \( f_2 \in F_2 \) is non-root-specific,

6. if there is a rule with a left hand side root \( f_1 \in F_1 \) and/or a right hand side root \( f_1' \in F_1 \) then there is also a rule with a left hand side root \( f_2 \in F_2 \) and/or a right hand side root \( f_2' \in F_2 \), with \( f_1, f_2 \) and/or \( f_1', f_2' \) the \( R \)-related constructors to be identified,

7. if there is a rule with a left hand side root \( f_1 \in F_1 \) then the right hand side root is either in \( F_1 \) or non-specific.

Explanation. The root position identification transformation is the inverse of the root position distinguishing transformation, so it has to be possible to identify sets of pairs of constructors simultaneously. Pre-condition 2 is a necessary and self-evident requirement for pairs of constructors to be identifiable. The Pre-conditions
3, 4, and 5 guarantee that it is possible to reconstruct the origin of identified constructors out of the surrounding context. Pre-condition 6 and 7 is to guarantee that it is possible to derive a simulating rule set.

**Derivation of Vocabulary**

- \( C'' = C - F_1 - F_2 \cup F; \)
- \( S'' = S - F_1 \cup \{ f \in F \mid f_1 \in F_1 \cap S, f_1 Rf \}; \)
- \( \alpha'' = \alpha \uparrow \{ f_1 \leftarrow \perp, \quad f_2 \leftarrow \perp, \quad f \leftarrow \alpha(f_1) \mid \}
  \begin{align*}
  f_1 & \in F_1, \quad f_2 \in F_2, \quad f \in F, \\
  f_1 Rf, & f_2 Rf; \end{align*} \)
- \( \rho'' = \rho \uparrow \{ (f_1, i) \leftarrow \perp, \quad (f_2, i) \leftarrow \perp, \quad (f, i) \leftarrow \rho(f_1, i) \cup \rho(f_2, i), \quad (c, m) \leftarrow \rho(c, m) - \{ f_2 \} \cup \{ f \} \mid \}
  \begin{align*}
  c & \in C, \quad 1 \leq m \leq \alpha(c), \\
  f_1 & \in F_1, \quad f_2 \in F_2, \quad f \in F, \quad f_1 Rf, f_2 Rf, \\
  1 \leq i \leq \alpha(f_1), \\
  f_2 & \in \rho(c, m) \}; \)
- \( YV'' = \{ yv - F_1 - F_2 \cup \}
  \begin{align*}
  \{ f \in F \mid f_1 \in yv \cap F_1, f_1 Rf \} \cup \\
  \{ f \in F \mid f_2 \in yv \cap F_2, f_1 Rf \} \mid yv \in YV \}. \)

**Definition of Domain Mapping**

The description of the domain mapping associated with this transformation cannot be done with term rewriting systems for reasons mentioned earlier. A description with induction on term structure will be given of \( \phi : \mathcal{T}_\Sigma \rightarrow \mathcal{T}_{\Sigma'} \), using auxiliary function \( \phi' : \mathcal{R}_\Sigma \rightarrow \mathcal{R}_{\Sigma'} \) with:

- \( \phi(f_1(t_1, ..., t_{\alpha(f_1)})) = f(\phi'(t_1), ..., \phi'(t_{\alpha(f_1)})) \)
for all \( f_1 \in F_1, f \in F, f_1 Rf, \)

- \( \phi(c(t_1, ..., t_{\alpha(c)})) = c(\phi'(t_1), ..., \phi'(t_{\alpha(c)})) \)
  for all \( c \notin (F_1 \cup F_2), \)

- \( \phi'(f_2(t_1, ..., t_{\alpha(f_2)})) = f(\phi'(t_1), ..., \phi'(t_{\alpha(f_2)})) \)
  for all \( f_2 \in F_2, f \in F \) with \( f_2 Rf, \)

- \( \phi'(c(t_1, ..., t_{\alpha(c)})) = c(\phi'(t_1), ..., \phi'(t_{\alpha(c)})) \)
  for all \( c \notin (F_1 \cup F_2). \)

**Explanation.** The system is designed such that \( \phi \) is applied only to top-level terms and \( \phi' \) is applied only to sub-terms. Root occurrences of \( f \) are replaced by \( f_1 \) and internal occurrences of \( f \) by \( f_2 \).

**Derivation of Rewrite System**

(consequences for pattern's internal)

For each rule \( h = (l, r) \) in rulearea Do:

- Replace all occurrences of \( f_1 \in F_1 \) and \( f_2 \in F_2 \) in both \( l \) and \( r \) by \( f \in F \) for which \( f_1 Rf \) or \( f_2 Rf \);
- Replace all variables in both \( l \) and \( r \) the \( \mu \)-value of which contains \( f_2 \in F_2 \) with one with same \( \mu \)-value except that \( f_2 \) has been exchanged for \( f \in F \) for which \( f_2 Rf \).

End of For-iteration over rules.

**Explanation.** Let the assumptions be as stated in the beginning of this section. The consequences for the rules of a rewrite system for the original vocabulary are simple for this transformation. If left hand sides and/or right hand sides of the rules contain occurrences of the constructor \( f_2 \) at non-root positions then these occurrences can safely be replaced by constructor \( f \).

If left hand sides and/or right hand sides of the rules contain variables the \( \mu \)-value of which contains \( f_2 \) then these variables have to be replaced by variables with same \( \mu \)-value except that \( f_2 \) has been exchanged for \( f \).

If left hand sides or right hand sides of the rules contain occurrences of the constructor \( f_1 \) or \( f_2 \) at root-positions then the situation is easy in comparison with root side identification, thanks to the strong conditions for the root position identification transformation. It is not possible for a rule to restrict its operation to matching at root positions only, or to matching at non-root positions only, other than based on the identity of the root constructor of the terms to be rewritten. The identification which is performed in this transformation destroys the possibility to restrict operation of rules based on the identity of constructors. This means that the functionality of rules destined for operation at root positions and the functionality of rules destined for operation at non-root positions in the original system has to
be identical after the identification. This means in concrete terms for the derivation that if there is a rule:

\[ f_1(t_1, \ldots, t_n) \rightarrow f'_1(t_1, \ldots, t_n), \]

then there has to be a rule

\[ f_2(t_1, \ldots, t_n) \rightarrow f'_2(t_1, \ldots, t_n), \]

in the system as well, with \( f_1, f_2 \) two \( R \)-related constructors and \( f'_1, f'_2 \) two \( R \)-related constructors (possibly \( f_1 = f'_1 \) and \( f_2 = f'_2 \)). After identification the two rules become identical and the number of rules decreases because of the set structure in which the rules are embedded. Same arguments hold for the case in which left hand side roots are not members of \( F_1 \) and \( F_2 \) or the case in which right hand side roots are not members of \( F_1 \) and \( F_2 \).

### 6.10 Contraction

**Assumptions and Pre-conditions**

1. \( pa \) and \( ch \) are the constructors to be contracted into \( c \) if \( ch \) occurs at position \( p \) of \( pa \), with \( c \) a new constructor which does not already occur in \( C \) (figure 6.1),

2. \( ch \in \rho(pa, p) \),

3. \( pa \neq ch \).

**Explanation.** Pre-condition 3 is to eliminate a source of ambiguity. If two adjacent occurrences of one constructor are allowed to be contracted it is not clear what should be done with terms with three or more adjacent occurrences of this constructor. In practice it is virtually always possible to make this condition to hold for adjacent occurrences of this constructor by distinguishing this constructor one way or another.

**Derivation of Vocabulary**

- \( C'' = C \cup \{c\} \);
- \( S'' = \text{if } pa \in S \text{ then } S \cup \{c\} \text{ else } S \);
- \( \alpha'' = \alpha \cup \{c \leftarrow \alpha(pa) + \alpha(ch) - 1\} \);
- \( \rho'' = \rho \uparrow \{\langle pa, p \rangle \leftarrow \rho(pa, p) - \{ch\}, \langle an, i \rangle \leftarrow \rho(an, i) \cup \{c\} \} \)
\[ an \in C, 1 \leq i \leq \alpha(an), \]
\[ pa \in \rho(an, i) \}; \]

- \[ YV'' = \{ \text{if } pa \in yv \text{ then } yv \cup \{c\} \text{ else } yv \mid yv \in YV \}. \]

The transformation may introduce non-integratedness (definition 5.9) and non-satisfiability (definition 5.10) for the constructors \( pa \) and \( ch \). The conditions for this to happen are simple but repairing this non-integratedness and non-satisfiability has consequences for each element of the defining tuple, so the compact scheme for the derivation of the new vocabulary would become needlessly complicated if it was incorporated. The elimination of non-integratedness and non-satisfiability can be done with a separate transformation. In section 9.6.2 and 9.6.3 schemes for the disposal of non-integrated and non-satisfiable constructors will be presented. The conditions for \( pa \) and \( ch \) to become non-integrated or non-satisfiable will not only play a role in the description of these properties but also in the derivation of rewrite systems. These conditions will therefore be defined in a general way with named predicates.

Definition 6.1. Let \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) be a vocabulary with \( \{pa, ch\} \subseteq C \) and \( 1 \leq p \leq \alpha(pa) \). The predicates \( P1 \) and \( P2 \) are defined as follows:

- \( P1(pa, ch, p) \Leftrightarrow \rho(pa, p) = \{ch\}, \)
- \( P2(pa, ch, p) \Leftrightarrow \)
  \[ ch \notin S \wedge \forall c_x \in C, 1 \leq p_x \leq \alpha(c_x) \left[ ch \in \rho(c_x, p_x) \Rightarrow c_x = pa \wedge p_x = p \right] \]

The conditions for \( pa \) and \( ch \) to become non-integrated or non-satisfiable:

- \( pa \) becomes non-integrated \( \Leftrightarrow P1(pa, ch, p), \)
- \( ch \) becomes non-integrated \( \Leftrightarrow P2(pa, ch, p), \)
- \( pa \) becomes non-satisfiable at position \( p \) \( \Leftrightarrow P1(pa, ch, p). \)

Definition of Domain Mapping

\[ \{ \{pa(v_{1...j-1}, ch(v_{j...j+n-1}), v_{j+n...v_{m+n-1}}) \rightarrow c(v_{1...v_{m+n-1}}) \mid \]
\[ v_{1,...,v_{m+n-1}} \text{ are distinct variables} \]
\[ \} \}_{\alpha}. \]

Derivation of Rewrite System

(\textit{consequences for pattern's root side})
For each rule \( h = \langle c_{l(t_1,...,t_{\alpha(cl)}), c_r(u_1,...,u_{\alpha(cr)})} \rangle \) in rulearea Do:
If \( c_l = ch \lor c_r = ch \) Then
  If \( P\bar{P}(pa, ch, p) \) Then
    Remove \( h \) from rulearea.
  End of If-Then;
  Add a rule to rulearea which is derived from \( h \) by replacing the pattern
  \( c_x(t_1, \ldots, t_{\alpha(ex)}) \) (\( x = l \) or \( x = r \)) by:
  \( c(v_1, \ldots, v_{p-1}, t_1, \ldots, t_{\alpha(ex)}, v_{p+1}, \ldots, v_m) \)
  if \( c_x = ch \) and by:
  \( pa(v_1, \ldots, v_{p-1}, c_x(t_1, \ldots, t_{\alpha(d)}), v_{p+1}, \ldots, v_m) \)
  if \( c_x \neq ch \), with \( v_1, \ldots, v_{p-1}, v_{p+1}, \ldots, v_m \): distinct fresh variables which are
  maximally typed.
End of If-Then.
End of For-iteration over rules in rulearea;

(consequences for pattern's leaf side)
For each rule \( h = (l, r) \) in rulearea Do:
  Place a copy of \( h \) in an emptied workarea1;
  For each variable \( v \) in \( Var(l) \) Do:
    For each rule \( h' = (l', r') \) in workarea1 Do:
      If \( v \) occurs below \( pa \) at position \( p \) Then
        If \( \exists c_x \in \mu(v) [c_x \neq ch] \) Then
          Remove \( h' \) from workarea1.
        End If-Then;
      End If-Then;
      If \( ch \in \mu(v) \) Then
        Add a rule to workarea1 which is derived from \( h' \) by replacing
        occurrences
        \( pa(t_1, \ldots, t_{p-1}, v, t_{p+1}, \ldots, t_m) \)
        of \( v \) below \( c \) at \( p \) by:
        \( c(t_1, \ldots, t_{p-1}, v_1, \ldots, v_n, t_{p+1}, \ldots, t_m) \)
        and other occurrences of \( v \) by:
        \( ch(v_1, \ldots, v_n) \)
        with \( v_1, \ldots, v_n \): distinct fresh variables which are maximally typed.
      End If-Then;
    End For-iteration over rules in workarea1.
    End of For-iteration over variables in \( Var(l) \);
  Add rules in workarea1 to workarea2.
End of For-iteration over rules;
Set rulearea to workarea2;

(consequences for pattern's internal)
For each rule \( h = (l, r) \) in rulearea Do:
  Replace all combined occurrences of \( pa \) and \( ch \) in both \( l \) and \( r \) by an occurrence
  of \( c \);
Replace all variables in both \( l \) and \( r \) the \( \mu \)-value of which contains \( pa \) with one with same \( \mu \)-value except that \( c \) has been added to this \( \mu \)-value.

End of For-iteration over rules.

Explanations. Let the assumptions be as stated in the beginning of this section. Suppose rule \( h = (l, r) \) has to be transformed. If left hand side \( l \) and/or right hand side \( r \) of the rule contain combinations of \( pa \) and \( ch \) then the necessary action is straightforward: see comments after consequences for patterns internal. There are complications if \( ch \) occurs as root constructor of \( l \) and/or \( r \) and if \( pa \) occurs in \( l \) and/or \( r \) with a variable at position \( p \) because part of the combination that would be collapsed into one node by the transformation could be rewritten by these rules. In these complicating circumstances the following transformation actions may be possible:

i. the original rule has to be maintained or it has to be removed from the rule set, depending on certain conditions that will be given below,

ii. one or more derived rules (derived from the original rule) have to be added to the rule set, depending on certain conditions that will be given below.

Firstly we will consider what the consequences are of a \( ch \) constructor occurring at the root of \( l \) and/or \( r \). Secondly we will consider what the consequences are of a \( pa \) constructor occurring somewhere in \( l \) and/or \( r \) with a variable at position \( p \). Finally we will pay attention to combined occurrences of \( pa \) and \( ch \) in the patterns' internal.

Consequences for pattern's root side. If a \( ch \) constructor occurs at the root of \( l \) and/or \( r \) then obviously the original rule has to be maintained if and only if \( ch \) is allowed to occur at other positions than only below \( pa \) at \( p \), formally:

\[ h \text{ has to be maintained} \iff P2(pa, ch, p). \]

The rule that has to be added in these circumstances is the rule that can be derived from \( h = (l, r) \) as follows (recall that at least one of the roots of \( l \) or \( r \) equals \( ch \)). The root constructors that equal \( ch \) have to be replaced by \( c \) and the root that does not equal \( ch \) has to be extended at the root by an occurrence of constructor \( c \). At both sides of the rule there are now \( m - 1 \) empty operand positions that have to be filled in by something: \( m - 1 \) because the added constructor \( pa \) has \( m \) positions and only one is occupied by the \( ch \)-term and \( m - 1 \) because the replacing constructor \( c \) has \( m + n - 1 \) positions and only \( n \) are occupied by the \( n \) operands of the original constructor. These \( m - 1 \) positions have to be filled up with distinct fresh variables which are maximally typed, corresponding variables at left-right corresponding positions. A example to illustrate the derivation of a rule to be added. Suppose \( pa \) and \( ch \) are as in figure 6.1 and suppose moreover that:

\[ p = 3, \]
\[ \alpha(pa) = 4, \]
\[ \rho(pa, 3) = \{ch, f\}. \]

Suppose that we have to transform the rule:

\[ ch(\ldots) \rightarrow f(\ldots). \]

The derived rule will look like:

\[ c(v_1, v_2, \ldots, v_3) \rightarrow pa(v_1, v_2, f(\ldots), v_3). \]

Consequences for pattern's leaf side. If a \( pa \) constructor occurs somewhere in \( l \) and/or \( r \) with a variable at position \( p \), then obviously the original rule has to be maintained if and only if other constructors than \( ch \) are allowed to match as root constructor against the variable in question. Formally:

\[ h \text{ has to be maintained } \iff \exists c_x \in \mu(v) \ [c_x \neq ch]. \]

Each distinct variable that occurs in \( l \) can occur below \( pa \) at \( p \). The appropriate action for these variables depends on the allowed root constructors for a match against these variables, i.e. it depends on the \( \mu \)-values of the variables that occur somewhere in \( l \) or in \( r \) (or both) under \( pa \) at \( p \).

If \( ch \in \mu(v) \) then \( v \) gives rise to a replacement action, if \( ch \neq pa \in \mu(v) \) (i.e. if there is at least one other constructor) then \( v \) gives rise to a maintenance action, if \( ch \in \mu(v) \) and \( ch \neq pa \in \mu(v) \) then \( v \) gives rise to a replacement and a maintenance action. This last case of replacement and maintenance will result in combinations if there is more than one variable with the “replacement and maintenance” action.

In case of “maintenance” the rule should be kept as it was. In case of “replacement” the variable occurrences of the variable \( v \) in question should be treated as follows both in \( l \) and in \( r \):

i. if \( v \) occurs below \( pa \) at \( p \) then \( pa \) should be replaced by \( c \), the operands of \( pa \) at other positions than position \( p \) should be moved to the \( p - 1 \) positions and the last \( m - p - 1 \) positions. The remaining \( n \) positions should be filled up with distinct fresh variables which are maximally typed,

ii. if \( v \) does not occur below \( pa \) or not at \( p \) then \( v \) should be replaced by a constructor \( ch \) with as operands the same \( n \) fresh variables as in action point i, and in the same order.

An example to illustrate the construction of a rule for the “replacement” case of variable \( v \). Suppose \( pa \) and \( ch \) are as in figure 6.1 and suppose moreover that:

\[ p = 3, \]
\[ \alpha(pa) = 4, \]
\[ \alpha(f) = 1, \]
\[ \rho(f, 1) = \{ch\}, \]
\[ \alpha(ch) = 3. \]

Suppose that we have to transform the rule:
\[ pa(v) \rightarrow f(v). \]

The "replacement" case will result in the rule:
\[ c(v_1, v_2, v_3) \rightarrow f(ch(v_1, v_2, v_3)). \]

This rule covers the functionality of the original rule completely because the "maintenance" case is empty (the original rule needs not to be maintained because there is no \( c \) in \( \mu(v) \) other than \( ch \)).

Consequences for pattern's internal. If left hand side \( l \) and/or right hand side \( r \) of the rule contain combinations of \( pa \) and \( ch \) then these combinations should simply be contracted into \( c \).

### 6.11 Expansion

**Assumptions and Pre-conditions**

1. \( c \) is the constructor to be expanded into \( pa \) and \( ch \), with \( pa \) and \( ch \) two new constructors which do not already occur in \( C \) (figure 6.2), the operand positions numbered \( p \) through \( p+n-1 \) of \( c \) will become subordinate to \( ch \), the remaining operand positions of \( c \) will become subordinate to \( pa \),

2. \( \alpha(c) = m. \)

**Derivation of Vocabulary**

- \( C'' = C - \{c\} \cup \{pa, ch\}; \)
- \( S'' = if \ c \in S \ then \ S - \{c\} \cup \{pa\} \ else \ S; \)
- \( \alpha'' = \alpha \uparrow \{c \leftarrow \bot, pa \leftarrow m - n + 1, ch \leftarrow n\}; \)
- \( \rho'' = \rho \uparrow \{(c, i) \leftarrow \bot, \)
  \[ \langle pa, j \rangle \leftarrow \rho(c, j), \]
  \[ \langle pa, p \rangle \leftarrow \{ch\}, \]
  \[ \langle pa, k \rangle \leftarrow \rho(c, k), \]
6.12. CORRECTNESS

\[ \langle c_{ch}, l \rangle \leftarrow \rho(c, l + p - 1) \mid \\
1 \leq i \leq m, \\
1 \leq j < p, \\
p + n \leq k \leq m, \\
1 \leq l \leq n \}; \]

- \( YV'' = \{ if \ c \in yv \ then \ yv - \{c\} \cup \{pa\} \ else \ yv \mid yv \in YV \}. \)

Definition of Domain Mapping

\[ \{\{c(v_1, ..., v_m) \rightarrow pa(v_1, ..., v_{j-1}, ch(v_j, ..., v_{j+n}), v_{j+n+1}, ..., v_m) \mid \]

\[ v_1, ..., v_n \ are \ distinct \ variables \]

\} \}_{\alpha}. \]

Derivation of Rewrite System

(\textit{consequences for pattern's internal})

\textbf{For each rule} \( h = (l, r) \) \textbf{in rulearea Do:}

- Replace all occurrences of \( c \) in both \( l \) and \( r \) by occurrences of the constructor \( pa \) and \( ch \) with the right operands of \( c \) below \( ch \) and the remaining operands of \( c \) below \( pa \), everything in the right order;
- Replace all variables in both \( l \) and \( r \) the \( \mu \)-value of which contains \( c \), with one with same \( \mu \)-value except that \( c \) has been exchanged for \( pa \).

End of For-iteration over rules.

\textit{Explanation.} The consequences of this transformation for a term rewriting system are simple: replace all occurrences of \( c \) in both patterns of the rule by a combination of \( pa \) and \( ch \) with the sub-terms of \( c \) distributed in the right way across the position of \( pa \) and \( ch \). No more actions necessary. No rule multiplications in this case.

6.12 Correctness

Each of the descriptions of each of the elementary transformations comprises three formal expressions that need to be proven correct: the derivation of the vocabulary, the definition of the domain mapping and the derivation of simulating rule sets. It will be discussed what aspects of these formal expressions need verification. Most of these verifications are rather trivial.

The transformations are actually defined by the formal expressions for the derivation of the vocabulary \textit{and} the definition of the domain mapping. Certain aspects of these defining components need a verification of partial correctness however. Each
derived vocabulary has to be a vocabulary as required by definition 5.1 and each domain mapping has to be a bijective function between each possible original vocabulary and each corresponding derived vocabulary. This last requirement implies that if a domain mapping is defined by a rewrite system then this rewrite system has to correspond to a function (the rewrite system has to terminate and has to have unique normal forms). These verifications are all very simple and are omitted therefore.

The derivation of simulating rewrite systems is something which is entirely determined by the definition of the new vocabulary and the definition of the domain mapping. Verification of the fact that the derived rewrite system is a step simulator of the original system will proceed in similar ways for each of the eight transformations. A general, informal correctness reasoning will be given, which applies to each of the derivation algorithms.

The effect of the various transformations on terms to be transformed is always local to the involved constructors and is also locally determined, the involved constructors being the constructors to be identified, distinguished, contracted, or expanded. In case of identification and distinguishing the scope of the transformation extends to the constructor immediately above the involved constructor or to the constructors immediately below the involved constructors. In case of contraction and expansion the scope of the transformation does not extend beyond the involved constructors. So the scope of each elementary transformation is always restricted to a local area of at most two levels of constructors. This means that in the derivation of a simulating rewrite system attention needs to be paid only to occurrences of involved constructors inside rule patterns and possible occurrences of involved constructors at the bound of pattern occurrences.

If a local area around the involved constructors of one transformation is entirely inside the bounds of one pattern, the action to maintain the original functionality of the rule in the new domain is straightforward. The involved constructors simply have to be identified, distinguished, contracted, or expanded. This kind of action is always described in the clauses consequences for pattern's internal of the algorithms.

If a local area around the involved constructors of one transformation can occur on the bounds of a pattern occurrence, which means partly inside and partly outside the pattern occurrence, then the situation is more complicated and may give rise to rule multiplications. Rule multiplications are necessary when the operation of one single rule cannot be simulated by one single rule in the new domain; two or sometimes even more rules are necessary in that case. The reason for this is quite simple. If an affected area around the involved constructors of a transformation can occur partly inside and partly outside a pattern occurrence then the exact shape of this local area depends on the context in which the pattern matches. Making the right decision for the transformation of the local area (replacing or renaming certain constructors of that local area) is only possible if the local area is entirely determined. The only way to get local areas which are entirely determined is to extend the patterns in such a way that the local areas are entirely inside the pattern. Because contexts in which patterns can match vary in most situations, one pattern
has to be extended in different ways to match different surrounding contexts. This means that one pattern possibly gives rise to two or more patterns, and one rule possibly to two or more rules. The occurrences of local areas on the root side border of the pattern (which means that the root constructor of the pattern is part of the local area) is covered by the clauses *consequences for pattern's root side* of the algorithms. The occurrences of local areas on the leaf side border of the pattern (which means that a constructor immediately above a variable and that variable position are part of the local area) is covered by the clauses *consequences for pattern's leaf side* of the algorithms.

It is not difficult to see that the actions which are described in the algorithms for the various occurrence cases of local areas are right, i.e. preserve the rewrite relation from the original domain to the simulating domain. Correctness of the entire algorithm follows now from the fact that the effects of the transformations are local around the involved constructors and that the case analysis is exhaustive, i.e. a local area occurs either entirely inside a pattern, or at the root side border of the pattern, or at the leaf side border of the pattern. Note that if a local area occurs entirely outside the pattern the operation of the rule is not affected because of the locality of scope.
Chapter 7

Semantic Aspects of Transformations

7.1 Introduction

In this chapter it will be shown that the elementary transformations of chapter 6 correspond semantically to fundamental mechanisms of deriving operations from other operations: fundamental abstraction mechanisms. It will also be shown how to reinterpret the operations of a given semantics to obtain a valid semantics for a transformed system. This chapter is meant to show that the elementary transformations of chapter 6 can easily be harmonized with algebraic semantics as presented in section 2.4. The formalization of this harmonization: the domain homomorphism concept, states that the semantics of closed terms (ground terms) is not affected by the domain mappings associated with the elementary transformations of chapter 6, or can at least be recovered by a single function.

In section 7.2 the relation between the various elementary transformations and fundamental abstraction mechanisms will be discussed, in section 7.3 the description of the elementary transformations in terms of operand and result types of operators in a signature will be presented. The derivation processes as described in section 7.3 may introduce certain kinds of redundancies in the type systems. Section 7.4 is about simplifications of type systems which can remove these redundancies. Finally in section 7.5 the reinterpretation will be given of operations of an algebraic interpretation of a signature (an algebra), based on the domain homomorphism concept, which is a generalization of the algebraic homomorphism notion, and based on the type system modifications of section 7.3.

The construction of type systems with certain desired properties in section 7.3 and the construction of domain homomorphic algebras for each of the elementary transformations in section 7.5 may seem a bit tricky. The practical value of these constructions is questionable indeed. The intention is only to show that the elementary transformations, which are mere syntactical transformations, do not conflict with algebraic semantics.
7.2 Fundamental Abstraction Mechanisms

The identification transformations (section 6.7, 6.8, 6.9) correspond to generalizations at the semantic level. Two (or more) constructors representing different operations, are replaced by one constructor, representing a more general operation. In algebraic terms this is a generalization of the two operations resulting in a new operation which can be used in a set of contexts which is the union of the contexts of the original operations. Generalization is a so called fundamental abstraction mechanism [31].

The distinguishing transformations (section 6.4, 6.5, 6.6) correspond to specializations at the semantic level. One constructor, representing an operation, is replaced by two (or more) constructors, which represent different operations. This is in algebraic terms a specialization of the operation with regard to distinct contexts in which this operation can occur. Specialization is the inverse operation of generalization.

The contraction transformation (section 6.10) corresponds to (functional) composition at the semantic level. Two vertically related constructors, representing two consecutive operations, are replaced by one constructor, representing the composition of the original operations. This is in algebraic terms operational composition which results in more complex operations that can be used instead of certain combinations of simpler operations. Composition is a so called fundamental abstraction mechanism [31].

The expansion transformation (section 6.11) corresponds to (functional) decomposition at the semantic level. One constructor, representing an operation, is replaced by two vertically related constructors which together (by composition) represent the original operation. This is in algebraic terms operational decomposition which results in a more refined description of complex operations. Decomposition is the inverse operation of composition.

7.3 Consequences for Operand and Result Types

7.3.1 Introduction

The elementary transformations of chapter 6 were defined in terms of vocabularies and the primitives of the retrenched vocabulary formalism, especially the restriction function $\rho$. It will be discussed how the elementary transformation can be defined in terms of modifications in a type system for a plain order sorted signature. The main differences between the retrenched vocabulary formalism and the order sorted signature formalism were (section 5.4):

- a shortcut between operand and result types of operators, resulting in a restriction function for constructors,

- a restriction of the set of constructors which is allowed to act as root constructor of well-formed terms.
7.3. CONSEQUENCES FOR OPERAND AND RESULT TYPES

The first item is only a superficial, syntactical difference. The second item is a structural difference and defining the elementary transformations in terms of order sorted signatures requires signatures which are enriched with a primitive for the restriction of certain operators at the root position of well-formed terms. For that reason the order sorted signature notion will be extended with a specially designated root type $s_0$, which prescribes the result type of root operators of well-formed terms. In the rest of this section order sorted signatures are assumed to be 5-tuples $(S, Rd ecs, Od ecs, Vd ecs, s_0)$, with $s_0 \in S$ the specially designated root type. The corresponding modifications in the definition of well-formed terms are left as an exercise to the reader (like with the definitions 5.5 and 5.7 a definition with two levels is necessary: one level that takes care of type violations between adjacent constructors and one level for the exclusion of improperly typed root operators, and like with the definitions 5.5 and 5.7 the term set without restrictions for the root operator can be identified with the symbol $R$ and the term set with restrictions for the root operator can be identified with the symbol $T$). Order sorted signatures with additional prescription for the type of entire terms will be called extended order sorted signatures in the rest of this chapter.

In the rest of this section a description will be given for each of the elementary transformations of how to express the intended modifications in the term structure in terms of types of operators and relations between types of an order sorted signature. It is assumed that the signatures are plain, i.e. they do not contain multiply declared operators. Each of the descriptions is informally shown correct, i.e. accomplishes the intended modification in the signature and the defined set of terms. The parts assumptions and pre-conditions of the descriptions to come, parallel the similar parts assumptions and pre-conditions in the descriptions of chapter 6. Each assumption or pre-condition of the lists in chapter 6 corresponds to an assumption or pre-condition in the list in this chapter, and in the same order.

In the forthcoming descriptions the operators $\overline{\top}$ (overbar) and $\overline{\overline{\top}}$ (double overbar) derive new types from given types, which do not already occur in the type systems of the signatures to be transformed. The binary operator $\uparrow$ returns the largest type of two given types (which must be related therefore), so:

$$s_1 \uparrow s_2 = \text{if } s_1 \leq s_2 \text{ then } s_2 \text{ else } s_1.$$

The functions $\text{Rtype}$, $\text{Otype}$, and $\text{Fits}$ (definition 2.2 and 2.3) are heavily used in the descriptions to come. Keep in mind that the pairs in $\text{Rdecs}$ are directed such that $(s_1, s_2) \in \text{Rdecs} \Rightarrow s_1 < s_2$.

### 7.3.2 Leaf Side Distinguishing

**Assumptions and Pre-conditions**

- $f$ is the operator to be distinguished in connection with position $p$ into $f_1$ and $f_2$ with $f_1$ and $f_2$ two new operators which do not already occur in $O_{\Sigma}$. 

CHAPTER 7. SEMANTIC ASPECTS OF TRANSFORMATIONS

- \langle s_1...s_n, s_r \rangle \) is the type specification of \( f \) and in particular: \( s_p \) is the type of operand position \( p \) of \( f \), i.e. \( s_p = \text{Otype}(f, p) \),

- \( \{ O_{ch,1}, O_{ch,2} \} \) is a partition of \( O_{ch} = \text{Fits}(s_p) \), i.e.
  \[
  \begin{align*}
  O_{ch} &= O_{ch,1} \cup O_{ch,2}, \\
  O_{ch,1} \cap O_{ch,2} &= \emptyset.
  \end{align*}
  \]

- If \( \text{Root}(l) \in O_{ch,i} \) and \( \text{Root}(r) \in O_{ch,j} \) for some \( i \in \{1, 2\} \) then \( \text{Root}(l) \not\in \text{Fits}(s_0) \),

- The rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with same functionality, section 9.2.5).

Derivation of Signature

- \( S'' = S \cup \{ \overline{t}, \overline{t} \mid t \in S, t \leq s_p \} \),

- \( R\text{decs}'' = R\text{decs} \cup \{ \overline{t}, t \mid t \in S, t \leq s_p \} \cup \)
  \[
  \{ (\overline{t_1}, \overline{t_2}), (\overline{t_1}, \overline{t_2}) \mid t_1 \leq s_p, t_2 \leq s_p, (t_1, t_2) \in R\text{decs} \}.
  \]

- \( O\text{decs}' = O\text{decs} - \{ \langle f, \langle s_1...s_n, s_r \rangle \rangle \} \cup \)
  \[
  \{ \langle f_1, \langle s_1...\overline{s_p}...s_n, s_r \rangle \rangle, \langle f_2, \langle s_1...\overline{s_p}...s_n, s_r \rangle \rangle \},
  \]

- \( O\text{decs}'' = \{ \text{if } g \in O_{ch,1} \)
  \begin{align*}
  \text{then } &\langle g, \langle t_1...t_n, \overline{t_r} \rangle \rangle \\
  \text{elif } g \in O_{ch,2} &\text{then } \langle g, \langle t_1...t_n, \overline{t_r} \rangle \rangle \\
  \text{else } &\langle g, \langle t_1...t_n, t_r \rangle \rangle \mid \langle g, \langle t_1...t_n, t_r \rangle \rangle \in O\text{decs}' \},
  \end{align*}

- \( V\text{decs}'' : \) countably infinite variables for each type, as required in definition 2.1,

- \( s''_0 = s_0 \).

Explanation

The type system is extended with two "shadow" structures which are copies of the type structure below and including \( s_p \): the operand type of operand position \( p \) of \( f \). One shadow structure consists of the \( \overline{a} \)-types, the other of the \( \overline{s} \)-types (figure 7.1 shows a type structure with two shadow structures; the meaning of the arrows is \( t \to t' \Leftrightarrow t > t' \)). Operators which have to be specialized to occur below \( f_1 \) are assigned an \( \overline{s} \)-corresponding result type and operators which have to be specialized
to occur below \( f_2 \) are assigned an \( \overline{\sigma} \)-corresponding result type. It is easy to see that by assigning \( \overline{\sigma}_p \) to \( f_1 \) at position \( p \) and \( \overline{\sigma}_p \) to \( f_2 \) at position \( p \), the operators below \( f \) at \( p \) are rightly divided between \( f_1 \) and \( f_2 \). Instead of following an \( s \)-chain to \( s_p \) it is possible to either follow an \( \overline{s} \)-chain to \( \overline{\sigma}_p \) or an \( \overline{\sigma} \)-chain to \( \overline{\sigma}_p \). It is also easy to see that operators from both classes can still occur at the remaining operand positions with type \( s_p \) or a super-type of \( s_p \) because both \( \overline{\sigma}_p \) and \( \overline{\sigma}_p \) are defined to be subtype of \( s_p \). The two shadow structures are necessary to guarantee that the operators from both classes can still occur at operand positions with a type which is a subtype of \( s_p \). These occurrences are possible because each \( \overline{s} \) and each \( \overline{\sigma} \) is defined to be a subtype of \( s \).

\[
\begin{align*}
&\text{figure 7.1: type structure with two lower shadow structures} \\
\end{align*}
\]

7.3.3 Root Side Distinguishing

Assumptions and Pre-conditions

- \( f \) is the operator to be distinguished into \( f_1 \) and \( f_2 \) with \( f_1 \) and \( f_2 \) two new operators which do not already occur in \( O_\Sigma \),
- \( \langle s_1 \ldots s_n, s_r \rangle \) is the type specification of \( f \),
- \( \{O_{pa,1}, O_{pa,2}\} \) is a partition of \( O_{pa} = \{ (f', p') \in O_\Sigma \times N \mid (f', \langle s'_1 \ldots s'_n \rangle) \in O_{dec}, s_r \leq s'_r \} \), i.e.
  - \( O_{pa,1} \cup O_{pa,2} = O_{pa} \),
  - \( O_{pa,1} \cap O_{pa,2} = \emptyset \),
- \( s_r \notin s_0 \).
The rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with same functionality, section 9.2.5).

**Derivation of Signature**

- \( S'' = S \cup \{ \bar{r}, \bar{t} \mid t \in S, s_r \leq t \} \),
- \( Rdecs'' = Rdecs \cup \{ (t, \bar{r}) \mid t \in S, s_r \leq t \} \cup \{ (\bar{t}_1, \bar{t}_2) \mid s_r \leq t_1, s_r \leq t_2, (t_1, t_2) \in Rdecs \} \),
- \( Odecs' = Odecs - \{ \langle f, \langle s_1 \ldots s_n, s_r \rangle \rangle \} \cup \{ \langle f_1, \langle s_1 \ldots s_n, \overline{s_r} \rangle \rangle, \langle f_2, \langle s_1 \ldots s_n, \overline{s_r} \rangle \rangle \} \),
- \( Odecs'' = \{ \langle g, \langle \text{if} \langle g, 1 \rangle \in Opa_1 \text{ then } \overline{t}_1 \text{ else } \overline{t}_1 \rangle, \langle g, \langle \text{if} \langle g, 2 \rangle \in Opa_2 \text{ then } \overline{t}_3 \text{ else } \overline{t}_3 \rangle, \ldots \rangle \mid \langle g, \langle t_1 \ldots t_n, t_r \rangle \rangle \in Odecs' \} \),
- \( \overline{s''} = s_0 \).

**Explanation**

The modifications in the type system for this transformation are like those for leaf side distinguishing but turned "upside down". Everything that was said about subtypes can be restated for super-types. The type system is extended with two "shadow" structures which are copies of the type structure above and including \( s_r \): the result type of \( f \). One shadow structure consists of the \( \overline{s} \)-types, the other of the \( \overline{\overline{s}} \)-types (figure 7.2 shows a type structure with two shadow structures; the meaning of the arrows is \( t \rightarrow t' \leftrightarrow t > t' \)). Operators which have to be specialized to occur above \( f_1 \) are assigned an \( \overline{s} \)-corresponding type at position \( p \) and operators which have to be specialized to occur above \( f_2 \) are assigned an \( \overline{\overline{s}} \)-corresponding type at position \( p \). It is easy to see that by assigning \( \overline{\overline{s}} \) as result type of \( f_1 \) and \( \overline{s} \) as result type of \( f_2 \) the operators "above" \( f \) are rightly divided between \( f_1 \) and \( f_2 \). Instead of following an \( s \)-chain from \( s_r \) upwards it is possible to either follow an \( \overline{s} \)-chain from \( \overline{s} \) or an \( \overline{\overline{s}} \)-chain from \( \overline{\overline{s}} \). It is also easy to see that operators from both classes can still occur above the remaining result types \( s_r \) or subtypes of \( s_r \) because both \( \overline{s} \) and \( \overline{\overline{s}} \) are defined to be super-type of \( s_r \). The two shadow structures are necessary to guarantee that the operators from both classes can still occur above operators with result type which is a super-type of \( s_r \). These occurrences are possible because each \( \overline{\overline{s}} \) and each \( \overline{s} \) is defined to be a super-type of \( s \).
7.3. CONSEQUENCES FOR OPERAND AND RESULT TYPES

7.3.4 Root Position Distinguishing

Assumptions and Pre-conditions

- $F$ is the set of operators to be distinguished into operators from $F_1$ and $F_2$ with $F_1$ and $F_2$ two sets of new operators which do not already occur in $O_S$, and with $R \subseteq F \times (F_1 \cup F_2)$ the relation which fixes the correspondence between operators and their replacements, i.e. $|F| = |F_1| = |F_2| = n$ and $f_i R f_{i+1}$ and $f_i R f_{2i}$, with $f_i \in F$, $f_{i+1} \in F_1$, $f_{2i} \in F_2$ for all $1 \leq i \leq n$.

- Each $f \in F$ is non-specific (section 6.3.2),

- The rule set contains no collapsing rules (this restriction is no real limitation since collapsing rules can easily be replaced by sets of non-collapsing rules with same functionality, section 9.2.5).

- If there are rules with a right hand side root in $F$ then the root of the left hand side is either in $F$ or specific (section 6.3.2).

- For all $1 \leq i \leq n$: $\langle s_{1i}, s_{mi}, s_{ri} \rangle$ is the type specification of $f_i \in F$.

Derivation of Signature

The following scheme can be used for root position distinguishing the set of operators $F$ into $F_1$ and $F_2$.

- $S'' = S \cup \{\bar{i}, \bar{t} | t \in S, 1 \leq i \leq n, s_{ri} \leq t\}$,
CHAPTER 7. SEMANTIC ASPECTS OF TRANSFORMATIONS

- $R_{decs''}$ = $R_{decs}$ $\cup$ \{(t, $\overline{t}$), (t, $\overline{t}$) | $t \in S, 1 \leq i \leq n, s_{ri} \leq t\} \cup$
  \{(t_1, t_2), (t_1, t_2) | 1 \leq i \leq n, s_{ri} \leq t_1, s_{ri} \leq t_2, (t_1, t_2) \in R_{decs}\},

- $O_{decs'}$ = $O_{decs}$ $-$ \{(f_{i1}, (s_{1i}, ..., s_{mi}, s_{ri})) | 1 \leq i \leq n\} $\cup$
  \{(f_{i1}, (s_{1i}, ..., s_{mi}, \overline{s_{ri}})), (f_{i1}, (s_{1i}, ..., s_{mi}, \overline{s_{ri}})) | 1 \leq i \leq n\}

- $O_{decs''}$ = \{(g, \{if $\exists 1 \leq i \leq n [s_{ri} \leq t_1] then \overline{t_1} else t_1$
  
  $\vdots$

  if $\exists 1 \leq i \leq n [s_{ri} \leq t_m] then \overline{t_m} else t_m,$

  $t_r$

  $\}) \} | (g, \langle t_1..., t_m, t_r \rangle) \in O_{decs'}\},$

- $V_{decs''}$ : countably infinite variables for each type, as required in definition 2.1,

- $s_0''$ = $\overline{s_0}$.

Explanation

The strategy for root position distinguishing is virtually the same as for root side distinguishing. Two "shadow" structures are introduced for the type structures above each $s_{ri}$: the result types of the operators to be distinguished. Instead of modifying the operand types of constructors in $O_{pa,1}$, depending on if the constructor-position pair is in $O_{pa,1}$ or in $O_{pa,2}$, all operand types $s_p$ at which some $f_i$ can occur are replaced by $\overline{s_p}$ (because if an operator occurs below another operator, it certainly does not occur at the root position) and $s_0$ is replaced by $\overline{s_0}$ (because $s_0$ prescribes the result type of operators occurring at the root position). The scheme is, like the transformation demands, designed for the simultaneous distinguishing of sets of operators. It is easy to see that each operator to be distinguished can use the same shadow type structures, so the $\overline{\tau}$- and $\overline{\tau'}$-operators can be the same for all indexes $i$.

7.3.5 Leaf Side Identification

Assumptions and Pre-conditions

- $f_1$ and $f_2$ are the operators to be identified in connection with operand position $p$ into $f$ with $f$ a new operator which does not already occur in $C$,

- The type specification of $f_1$ is $\langle s_{11}, ..., s_{1n}, s_{1r} \rangle$ and the type specification of $f_2$ is $\langle s_{21}, ..., s_{2n}, s_{2r} \rangle$ and in particular: $s_{1p} = O_{type}(f_1, p)$ and $s_{2p} = O_{type}(f_2, p)$.

- $\alpha(f_1) = \alpha(f_2),$

- $Fits(s_{1p}) \cap Fits(s_{2p}) = \emptyset,$
7.3. CONSEQUENCES FOR OPERAND AND RESULT TYPES

- \( \forall 1 \leq i \leq \alpha(f_1), i \neq p \left[ \text{Fits}(Otype(f_1, i)) = \text{Fits}(Otype(f_2, i)) \right] \),
- \( \forall op \in O_S, 1 \leq i \leq \alpha(op) \left[ s_{1r} \leq Otype(op, i) \Leftrightarrow s_{2r} \leq Otype(op, i) \right] \).

Derivation of Signature

- \( S'' = S \cup \{ \langle \overline{t}, t \rangle | t \in S, t \leq s_{1p} \} \cup \{ \langle \overline{t}, t \rangle | t \in S, t \leq s_{2p} \} \cup \{ ss \} \),
- \( Rdecs'' = Rdecs \cup \{ \langle \overline{t}, t', \overline{t''}, t'' \rangle | t', t'' \in S, t' \leq s_{1p}, t'' \leq s_{2p} \} \cup \langle \overline{t}, t' \rangle | t, t' \leq s_{1p}, \langle t, t' \rangle \in Rdecs \} \cup \langle \overline{t}, t' \rangle | t, t' \leq s_{2p}, \langle t, t' \rangle \in Rdecs \} \cup \{ \langle s_{1p}, ss \rangle, \langle s_{2p}, ss \rangle \}, \)
- \( Odecs' = Odecs - \{ \langle f_1, \langle s_{11}, \ldots, s_{1n}, s_{1r} \rangle \rangle, \langle f_2, \langle s_{21}, \ldots, s_{2n}, s_{2r} \rangle \rangle \} \cup \{ \langle f, \langle s_{11} \uparrow s_{21}, \ldots, s_{1p-1} \uparrow s_{2p-1}, ss, s_{1p+1} \uparrow s_{2p+1}, \ldots, s_{1n} \uparrow s_{2n}, s_{1r} \uparrow s_{2r} \rangle \rangle \} \)
- \( Odecs'' = \{ \text{if } g \in \text{Fits}(s_{1p}) \text{ then } \langle g, \langle t_1 \ldots t_n, \overline{t_r} \rangle \rangle \}
- \{ \text{elif } g \in \text{Fits}(s_{2p}) \text{ then } \langle g, \langle t_1 \ldots t_n, \overline{t_r} \rangle \rangle \}
- \{ \text{else } \langle g, \langle t_1 \ldots t_n, t_r \rangle \rangle \} \}
- \{ \langle g, \langle t_1 \ldots t_n, t_r \rangle \rangle \} \in Odecs' \}, \)

- \( Vdecs'' : \text{countably infinite variables for each type, as required in definition 2.1,} \)
- \( s''_0 = s_0. \)

Explanation

The type \( ss \) is a new type which also does not already occur in the type system of the signature to be transformed. The introduction of the \( \overline{t} \)-types and \( \overline{t'} \)-types is to guarantee that it is possible to construct an operation for \( f \) based on the operations associated with \( f_1 \) and \( f_2 \). The requirements are that \( \text{Fits}(s_{1p}) \cap \text{Fits}(s_{2p}) = \emptyset \); this does however not imply that \( s_{1p}^A \cap s_{2p}^A = \emptyset \) in each interpretation. Because there is no semantic requirement that \( f_1^A \) and \( f_2^A \) be equal on overlapping parts of their domains, we have to "tell" \( f^A \) which of the two possibilities the new "identified" operation should be taken on the overlapping part of both domains.

Since \( ss \) is defined to be a super-type of both \( s_{1p} \) and \( s_{2p} \), any operator which is allowed below \( f_1 \) or \( f_2 \) at \( p \) is allowed below \( f \) at \( p \). The construction of the new type specification for \( f \) with the operator \( \uparrow \) is because two types \( s' \) and \( s'' \) need not to be the same if \( \text{Fits}(s') = \text{Fits}(s'') \), though it is at least true that \( s' \leq s'' \) or \( s'' \leq s' \). Also \( s_{1r} \uparrow s_{2r} \) is a correct result type because the last pre-condition implies that at positions at which operators with result type \( s_{1r} \) or \( s_{2r} \) are allowed, operators with result type \( s_{1r} \uparrow s_{2r} \) are allowed as well.
7.3.6 Root Side Identification

Assumptions and Pre-conditions

- \( f_1 \) and \( f_2 \) are the operators to be identified into \( f \) with \( f \) a new operator which does not already occur in \( C \),

- the type specification of \( f_1 \) is \( \langle s_{11}...s_{1n}, s_{1r} \rangle \) and the type specification of \( f_2 \) is \( \langle s_{21}...s_{2n}, s_{2r} \rangle \),

- \( \alpha(f_1) = \alpha(f_2) \),

- \( \forall 1 \leq i \leq \alpha(f_1) \) \( \{ \text{Fits(\text{Otype}(f_1, i))} = \text{Fits(\text{Otype}(f_2, i))} \} \),

- \( \forall \text{op} \in O_\Sigma, 1 \leq i \leq \alpha(\text{op}) \) \( \{ s_{1r} \leq \text{Otype}(\text{op}, i) \Leftrightarrow s_{2r} \leq \text{Otype}(\text{op}, i) \} \).

Derivation of Signature

- \( S'' = S \cup \{ \bar{t} \mid s_{1r} \leq t \} \cup \{ \bar{s} \mid t \in S, s_{2r} \leq t \} \cup \{ss\}, \)

- \( R\text{decs}'' = R\text{decs} \cup \{ \langle t, \bar{t} \rangle \mid t \in S, s_{1r} \leq t \} \cup \{ \langle t, \bar{s} \rangle \mid s_{2r} \leq t \} \cup \)

\( \{ \langle \bar{t}, \bar{t}' \rangle \mid s_{1r} \leq t, t', \langle t, t' \rangle \in R\text{decs} \} \cup \)

\( \{ \langle \bar{t}, \bar{s} \rangle \mid s_{2r} \leq t, t', \langle t, t' \rangle \in R\text{decs} \} \cup \)

\( \{ \langle \bar{s}, \bar{s}_{1r} \rangle, \langle s, \bar{s}_{2r} \rangle \} \),

- \( O\text{decs}' = O\text{decs} - \{ \langle f_1, \langle s_{11}...s_{1n}, s_{1r} \rangle \rangle, \langle f_2, \langle s_{21}...s_{2n}, s_{2r} \rangle \rangle \} \cup \)

\( \{ \langle f, \langle s_{11} \uparrow s_{21}...s_{1n} \uparrow s_{2n}, s_{2r} \rangle \rangle \} \),

- \( O\text{decs}'' = \{ \langle g, \langle \mid \text{if } s_{1r} \leq t_1 \text{ then } \bar{t}_1 \text{ else } s_{2r} \leq t_1 \text{ then } \bar{t}_1 \text{ else } t_1 } \)

\( \vdots \)

\( \text{if } s_{1r} \leq t_n \text{ then } \bar{t}_n \text{ else } s_{2r} \leq t_n \text{ then } \bar{t}_n \text{ else } t_n, \)

\( \bar{t}_r \)

\( \rangle \rangle \mid \langle g, \langle t_1...t_n, t_r \rangle \rangle \in O\text{decs}' \} \),

- \( V\text{decs}'' \): countably infinite variables for each type, as required in definition 2.1,

- \( s_{0}'' = s_{0} \).
7.3. CONSEQUENCES FOR OPERAND AND RESULT TYPES

Explanation

The type $ss$ is a new type which does not already occur in the type system of the signature to be transformed. The introduction of the type $ss$ and the $\overline{\tau}$-types and $\overline{\overline{\tau}}$-types is to enable operators which occur above $f$ to decide between the results of $f_1$ and $f_2$. Recall that the identity of the operator directly above $f$ determines whether $f$ is a representative of $f_1$ or $f_2$. Therefore $f$ has to present the results of both $f_1$ and $f_2$ in a composed form to operators which can occur above it. Because of the subtype-subset requirement of definition 2.28, new super-types of $ss$ have to be introduced: the $\overline{\tau}$-types and $\overline{\overline{\tau}}$-types.

Since $ss$ is defined to be a subtype of both $\overline{s_{1r}}$ and $\overline{s_{2r}}$ and since every operand type which is a super-type of $s_{1r}$ or $s_{2r}$ is replaced by its associated $\overline{\tau}$-type or $\overline{\overline{\tau}}$-type, $f$ is allowed at the right operand positions. The construction of the operand types for $f$ with the operator $\uparrow$ is because two types $s'$ and $s''$ need not to be the same if $Fts(s') = Fts(s'')$, though it is at least true that $s' \leq s''$ or $s'' \leq s'$.

7.3.7 Root Position Identification

Assumptions and Pre-conditions

- $F_1$ and $F_2$ are the sets of operators to be identified into operators from $F$ with $F$ a set of new operators which do not already occur in $O_{\Sigma}$, and with $R \subseteq (F_1 \cup F_2) \times F$ the relation which fixes the correspondence between operators and their replacements, i.e. $|F_1| = |F_2| = |F| = n$ and $f_{1i}Rf_{1i}, f_{2i}Rf_{1i}$, with $f_{1i} \in F_1, f_{2i} \in F_2, f_i \in F$ for all $1 \leq i \leq n$.

- the type specification of $f_{1i}$ is $\langle s_{11i}, \ldots, s_{1ni}, s_{1ri} \rangle$ and the type specification of $f_{2i}$ is $\langle s_{21i}, \ldots, s_{2ni}, s_{2ri} \rangle$.

- $\forall f_{1i} \in F_1, f_{2i} \in F_2, f_i \in F [f_{1i}Rf_{1i} \land f_{2i}Rf_{1i} \Rightarrow \alpha(f_{1i}) = \alpha(f_{2i})]$.

- Each $f_{1i} \in F_1$ is root-specific,

- Each $f_{2i} \in F_2$ is non-root-specific,

- If there is a rule with a left hand side root $f_1 \in F_1$ and/or a right hand side root $f'_1 \in F_1$ then there is also a rule with a left hand side root $f_2 \in F_2$ and/or a right hand side root $f'_2 \in F_2$, with $f_{1i}, f_{2i}$ and/or $f'_{1i}, f'_{2i}$ the $R$-related constructors to be identified,

- If there is a rule with a left hand side root $f_1 \in F_1$ then the right hand side root is either in $F_1$ or non-specific.

Derivation of Signature
The following scheme can be used for root position identifying the sets of operators $F_1$ and $F_2$ into $F$.

The type $ss_i$ is a new type which does not already occur in the type system of the signature to be transformed for each $1 \leq i \leq n$.

- $S'' = S \cup \{i | 1 \leq i \leq n, s_{1ri} \leq t\} \cup \{t | 1 \leq i \leq n, s_{2ri} \leq t\} \cup \{ss_i\}$,
- $Rdecs'' = Rdecs \cup \{(t, i) | 1 \leq i \leq n, s_{1ri} \leq t\} \cup \{(t, t') | 1 \leq i \leq n, s_{2ri} \leq t\} \cup \{(i, t') | 1 \leq i \leq n, t, t' \in Rdecs\} \cup \{(t, t') | 1 \leq i \leq n, s_{2ri} \leq t, t, t' \in Rdecs\} \cup \{(ss_i, s_{1ri}), (ss_i, s_{2ri}) | 1 \leq i \leq n\}$,
- $Odecs'' = Odecs - \{(f_{1i}, s_{1ri}..., s_{1mi}, s_{1ri}), (f_{2i}, s_{2ri}..., s_{2mi}, s_{2ri}) | 1 \leq i \leq n\} \cup \{(f_{i}, s_{1ri}\uparrow s_{1ri}..., s_{1mi}\uparrow s_{2mi}, ss) | 1 \leq i \leq n\}$,
- $Odecs'' = \{\langle g, \epsilon | \exists 1 \leq i \leq n [s_{2ri} \leq t_1] \text{ then } \bar{t}_1 \text{ else } t_1 \}
  \vdots
  \text{if } \exists 1 \leq i \leq n [s_{2ri} \leq t_m] \text{ then } \bar{t}_m \text{ else } t_m,
  t_r \}
  \langle g, \langle t_1...t_m, t_r \rangle \rangle \in Odecs''\}$,
- $Vdecs''$: countably infinite variables for each type, as required in definition 2.1,
- $s''_0 = s_0$.

**Explanation**

The strategy for root position identification is virtually the same as for root side identification. The differences are that root position identification is defined for sets of pairs of operators, which means that the given scheme may have to be applied repeatedly. Another difference is that it is assumed that operators from $F_1$ cannot occur at operand positions of operands and operators from $F_2$ cannot occur at root-positions of well-formed terms. This allows us to omit one branch in the conditional expressions of the definition of $Odecs''$. 
7.3. CONSEQUENCES FOR OPERAND AND RESULT TYPES

7.3.8 Contraction

Assumptions and Pre-conditions

- $pa$ and $ch$ are the operators to be contracted into $c$ if $ch$ occurs at position $p$ of $pa$, with $c$ a new operator which does not already occur in $C$ (figure 6.1),

- $s_p = Otype(pa, p), s_R = Rtype(ch),$

- $ch \in Fits(s_p),$

- $pa \neq ch.$

Derivation of Signature

First alternative:

- $S'' = S \cup \{ \bar{t} \mid t \leq s_p \},$

- $Rdecs'' = Rdecs \cup \{ \langle \bar{t}, t \rangle \mid t \leq s_p \} \cup \{ \langle \bar{t}_1, \bar{t}_2 \rangle \mid t_1 \leq s_p, t_2 \leq s_p, (t_1, t_2) \in Rdecs \},$

- $Odecs' = Odecs - \{ \langle pa, \langle s_1 \ldots s_p \ldots s_n, s_r \rangle \rangle \} \cup \{ \langle pa, \langle s_1 \ldots s_p \ldots s_n, s_r \rangle \rangle, \langle c, \langle Otype(pa, 1) \ldots Otype(pa, p - 1), Otype(ch, 1) \ldots Otype(ch, n), Otype(pa, p + 1) \ldots Otype(pa, m), Rtype(pa) \rangle \rangle \} ,$

- $Odecs'' = \{ \text{if} \, op \neq ch \land s_r \leq s_p$
  then $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle$
  else $\langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle \mid \langle op, \langle s_1 \ldots s_n, s_r \rangle \rangle \in Odecs' \},$

- $Vdecs''$: countably infinite variables for each type, as required in definition 2.1,

- $s_0'' = s_0.$

Second alternative:

- $S'' = S \cup \{ \bar{t} \mid s_R \leq t \},$

- $Rdecs'' = Rdecs \cup \{ \langle t, \bar{t} \rangle \mid s_R \leq t \} \cup \{ \langle \bar{t}_1, \bar{t}_2 \rangle \mid s_R \leq t_1, s_R \leq t_2, (t_1, t_2) \in Rdecs \},$

- $Odecs' = Odecs - \{ \langle ch, \langle s_1 \ldots s_n, s_R \rangle \rangle \} \cup \{ \langle ch, \langle s_1 \ldots s_n, s_R \rangle \rangle, \langle c, \langle Otype(pa, 1) \ldots Otype(pa, p - 1), Otype(ch, 1) \ldots Otype(ch, n), Otype(pa, p + 1) \ldots Otype(pa, m), Rtype(pa) \rangle \rangle \} ,$
CHAPTER 7. SEMANTIC ASPECTS OF TRANSFORMATIONS

- $Odec's'' = \{\ \text{if } op \neq pa \wedge s_R \leq s_p \ \\
  \text{then } \langle op, \langle s_1 \ldots s_p \ldots s_n, s_r \rangle \rangle \ \\
  \text{else } \langle op, \langle s_1 \ldots s_p \ldots s_n, s_r \rangle \rangle \mid \langle op, \langle s_1 \ldots s_p \ldots s_n, s_r \rangle \rangle \in Odec's \}$,

- $Vdec's$: countably infinite variables for each type, as required in definition 2.1,

- $s''_0 = s_0$.

Explanation

The transformation requires the addition of a new operator $c$ with appropriate result and operand types, and a modification in the type system such that $ch$ is no longer allowed to occur below $pa$ at position $p$, while $ch$ is still allowed to occur on any other operand position and while any other operator is still allowed to occur below $pa$ at position $p$. The addition of the new operator is straightforward, the denial of $ch$ at position $p$ of $pa$ is more difficult because it should not have any effect on other occurrence permissions. There are two alternative solutions that are presented both because they are in principle equivalent. The first alternative is the approach with a "shadow" structure for all types below and including $s_p = Otype(pa, p)$, the second alternative is the approach with a "shadow" structure for all types above and including $s_R = Rtype(ch)$.

First alternative: it is easy to see that $ch$ can no longer occur below $pa$ at position $p$; $Otype(pa, p) = s_p$, which is not a super-type of $Rtype(ch) = s_R$. All other operators which were allowed below $pa$ at $p$ can still occur there because the result types $s_r$ of each of these operators is redefined to be $s_R$. The permission of these operators for other operand positions is not affected because $s < s_R$ for all involved types $s$.

Second alternative: it is easy to see that $ch$ can no longer occur below $pa$ at position $p$; $Rtype(ch) = \bar{s}_R$, which is not a subtype of $Otype(pa, p) = s_p$. All other operators which could occur above $ch$ can still occur there because the operand types $s_p$ of each of these operators (at the particular positions) is redefined to be $\bar{s}_R$. The permission of these operators (at the particular positions) for other operators is not affected because $s < \bar{s}$ for all involved types $s$.

7.3.9 Expansion

Assumptions and Pre-conditions

- let $c$ be the operator to be expanded into $pa$ and $ch$, with $pa$ and $ch$ two new operators which do not already occur in $C$ (figure 6.2), the operand positions numbered $p$ through $p + n - 1$ of $c$ will become subordinate to $ch$, the remaining operand positions of $c$ will become subordinate to $pa$,

- $\alpha(c) = m$. 
7.4. SIMPLIFYING TYPE SYSTEMS

Derivation of Signature

- \( S'' = S \cup \{ss\} \),
- \( Rdecs'' = Rdecs \),
- \( Odecs'' = Odecs \setminus \{c, \langle s_1...s_m, s_r \rangle \} \cup \{\langle p_a, \langle s_1...s_{p-1}, s_{p+n}...s_m, s_r \rangle \rangle, \langle c_h, \langle s_{p...s_{p+n-1}}, ss \rangle \rangle \} \),
- \( Vdecs'' \): countably infinite variables for each type, as required in definition 2.1,
- \( s_0'' = \overline{s_0} \).

Explanation

The type \( ss \) is a new type, which does not already occur in the type system of the signature to be transformed. Since the new type \( ss \) is not related with any other type the only operator that can occur at position \( p \) below \( p_a \) is \( c_h \) and since the result type of \( p_a \) equals the result type of \( c \), \( p_a \) can occur wherever \( c \) can occur. The definition of the remaining operand types of \( p_a \) and \( c_h \) will attain the desired behaviour, as can be seen easily.

7.4 Simplifying Type Systems

For most of the transformations the consequences for the type systems are rather complex; only the expansion transformation is relatively simple. Large numbers of new types were introduced, which had to be related in a certain well-defined way. There are situations in which it is not necessary to introduce so many new types. The schemes are so complex because they have to be general. Consider for instance the leaf side distinguishing transformation and a situation in which all operators in \( O_{ch,1} \) have result type \( s_1 \) and all operators in \( O_{ch,2} \) have result type \( s_2 \). If \( s_1 \) and \( s_2 \) are not related, i.e. neither \( s_1 < s_2 \) nor \( s_2 < s_1 \), the desired effect can be attained by simply defining the operand type of \( f_1 \) at \( p \) to be \( s_1 \) and the operand type of \( f_2 \) at \( p \) to be \( s_2 \).

A general scheme will be presented to remove redundancy in order sorted type systems, which may have been introduced by conversion schemes for order sorted signatures. This type system simplification scheme can be applied to any order sorted signature, since it is often possible to simplify type systems that are designed as such. Note however that redundancy in type systems may have other purposes besides the operational functionality of the determination of which operator is allowed to occur at which operand position of which other operator. The redundancy may for instance have a documentary purpose.

There are two different ways in which order sorted type systems can be simplified while preserving the basic functionality of the system: determining which operator is allowed to act as which operand of what other operator.
1. Types which do not occur as operand or as result type of the operators of the signature can simply be discarded.

2. If for two different types \( s_1, s_2 \in S \) it holds that \( \text{Fits}(s_1) = \text{Fits}(s_2) \) then \( s_1 \) and \( s_2 \) can be identified, resulting in a decrease of the number of types and a reduction of the complexity of the type system.

**Scheme 7.1.** A scheme for the removal of unused types:

- \( S'' = \{ s_1, \ldots, s_n, s_r \in S \mid \langle \text{op}, \langle s_1 \ldots s_n, s_r \rangle \rangle \in \text{Odecs} \} \),
- \( \text{Rdecs}'' = \{ \langle s_1, s_2 \rangle \in S'' \times S'' \mid \langle s_1, s_2 \rangle \in \text{Rdecs}^+ \}^- \).

The other components of the signature: \( \text{Odecs} \) and \( \text{Vdecs} \), remain unchanged. Note that defining \( \text{Rdecs}'' \) directly in terms of \( \text{Rdecs} \) is difficult because types to be removed can play a “chaining” role in the declaration. This is for instance the case if \( \{ \langle s_1, s_2 \rangle, \langle s_2, s_3 \rangle \} \subseteq \text{Rdecs} \) and \( s_2 \) has to be removed; after this removal the pair \( \langle s_1, s_3 \rangle \), which was implied by transitivity in the original situation, has to be added. Difficult bookkeeping of “chaining” can be avoided by using the transitive closure of \( \text{Rdecs} \)\( \square \)

**Scheme 7.2.** A scheme for the identification of two types \( s_1 \) and \( s_2 \) into \( s \). Let \( x \) and \( y \) be two variables that range over the set of types and let the operator \( \overline{\Box} \) (overbar) be defined as \( \overline{x} = \text{if } x = s_1 \lor x = s_2 \text{ then } s \text{ else } x \) in the following scheme:

- \( S'' = \{ \overline{x} \mid x \in S \} \),
- \( \text{Rdecs}'' = \{ \langle \overline{x}, \overline{y} \rangle \mid \langle x, y \rangle \in \text{Rdecs}, \overline{x} \neq \overline{y} \} \),
- \( \text{Odecs}'' = \{ \langle \text{op}, \langle s_1 \ldots s_n, \overline{s_r} \rangle \rangle \mid \langle \text{op}, \langle s_1 \ldots s_n, s_r \rangle \rangle \in \text{Odecs} \} \),
- \( \text{Vdecs}'' = \{ \langle \text{var}, \overline{x} \rangle \mid \langle \text{var}, x \rangle \in \text{Vdecs} \} \).\( \square \)

If for two distinct types \( s_1, s_2 \in S \) it holds that \( \text{Fits}(s_1) = \text{Fits}(s_2) \) then \( s_1 \) and \( s_2 \) can be identified, resulting in a decrease of the number of types and a reduction of the complexity of the type system.

**Algorithm 7.3.** An algorithm for the simplification of type systems, based on the two schemes:

Remove unused types with scheme 7.1;
While there are types \( s_1, s_2 \in S \) with \( \text{Fits}(s_1) = \text{Fits}(s_2) \) and \( s_1 \neq s_2 \) Do:
  Identify \( s_1 \) and \( s_2 \) with scheme 7.2.
End While-Do.\( \square \)
7.5 Reinterpreting Operations

7.5.1 Introduction

After the description of the elementary transformations in terms of operand types and result types it is easy to define operations for the new signature in terms of operations for the original signature. The definition of the new operations will be done in such a way that the semantic value of the old term is not affected. A formalization of this property will be given by the domain homomorphism concept.

For the various new types new domains have to be introduced such that the requirements for these domains with associated types are valid, i.e. if $s_1 < s_2$ then $s_1^A \subseteq s_2^A$ (definition 2.28). This can be achieved by suitably defining the new domains and/or redefining the original domains.

Algebraic semantics of terms is that each constant corresponds to a value and each operator with an operation on values, resulting in a new value (section 2.4). This correspondence induces a correspondence between ground terms and values by the following recursive definition.

**Definition 7.4.** Correspondence between ground terms and values. Let $\Sigma = \langle S, Rdecs, Odecs, Vdecs, s_0 \rangle$ be an extended order sorted signature and let $A$ be a $\Sigma$-algebra. The correspondence between ground terms $t \in TG_\Sigma$ and their values is made explicit in the function $Val_A : RG_\Sigma \rightarrow UA$. The function is recursively defined as:

- $Val_A(t) = c^A$, if $t = c \in K$,
- $Val_A(t) = op^A(Val_A(t_1), ..., Val_A(t_n))$, if $t = op(t_1, ..., t_n)$.

Note that $Val_A$ is defined for the more general set $RG_\Sigma \supset TG_\Sigma$ because it has to be applied recursively to sub-terms.

The $Val$-function is a special case of the "denotation" function that can be found in literature on algebraic structures and algebraic specifications, and which is based on an assignment of values to the variables of the terms [20, 54]. We do not need to introduce this value assignment operation because we are dealing solely with ground terms. The reinterpretation of operations and the "transformation of models" is based on the domain homomorphism concept.

**Definition 7.5.** Let $\Sigma, \Sigma'$ be two (extended) order sorted signatures and $TG_{\Sigma'}$, and $\phi : TG_\Sigma \rightarrow TG_{\Sigma'}$ a surjective domain mapping. Let $A$ be a $\Sigma$-algebra and $A'$ a $\Sigma'$-algebra. A surjective function $\phi_{A',A} : UA' \rightarrow UA$ is said to be a domain homomorphism with regard to domain mapping $\phi$ if and only if:

$\forall t' \in TG_{\Sigma'} [\phi_{A',A}(Val_{A'}(t')) = Val_A(\phi(t'))]$. 

The domain homomorphism notion is a generalization of the conventional homomorphism notion. With the conventional homomorphism notion there is an equality requirement for the resulting value of each operator; with the domain homomorphism there is only an equality requirement for entire ground terms. The reason for this relaxation is that with domain mappings, at least with the domain mappings corresponding to our elementary transformations, a one-to-one correspondence between operators is lost. One operator in one term domain may correspond to two operators in the other term domain, or vice versa. The relation between homomorphisms and domain homomorphisms is subject of the following theorems.

**Theorem 7.6.** Let $\Sigma$ be an extended order sorted signature. Let $\mathcal{A}, \mathcal{A}'$ be $\Sigma$-algebras and $\phi_{\mathcal{A}', \mathcal{A}} : \mathcal{U}^{\mathcal{A}'} \rightarrow \mathcal{U}^{\mathcal{A}}$ a homomorphism from $\mathcal{A}'$ to $\mathcal{A}$. Now $\phi_{\mathcal{A}', \mathcal{A}} : \mathcal{U}^{\mathcal{A}'} \rightarrow \mathcal{U}^{\mathcal{A}}$ is a domain homomorphism with regard to domain mapping $id_{TG} : \mathcal{T}G_{\Sigma} \rightarrow \mathcal{T}G_{\Sigma}$ (the identity function on $\mathcal{T}G_{\Sigma}$).

**Proof with induction on the structure of terms.** Since $\phi = id_{TG}$ we have to prove that:

$$\forall t \in \mathcal{T}G_{\Sigma} [\phi_{\mathcal{A}', \mathcal{A}}(Val_{\mathcal{A}'}(t)) = Val_{\mathcal{A}}(t)].$$

Assume that the fact to be proven holds for all terms with size (number of constructors) less than that of $t$.

$$\phi_{\mathcal{A}', \mathcal{A}}(Val_{\mathcal{A}'}(t)) = (\text{definition } Val)$$

$$\phi_{\mathcal{A}', \mathcal{A}}(op^{\mathcal{A}'}(Val_{\mathcal{A}'}(t_1), ..., Val_{\mathcal{A}'}(t_n))) = (\phi_{\mathcal{A}', \mathcal{A}} \text{ is a homomorphism})$$

$$op^{\mathcal{A}}(\phi_{\mathcal{A}', \mathcal{A}}(Val_{\mathcal{A}'}(t_1)), ..., \phi_{\mathcal{A}', \mathcal{A}}(Val_{\mathcal{A}'}(t_n))) = (\text{induction hypothesis})$$

$$op^{\mathcal{A}}(Val_{\mathcal{A}}(t_1), ..., Val_{\mathcal{A}}(t_n)) = (\text{definition } Val)$$

$$Val_{\mathcal{A}}(t),$$

which we had to prove. $\square$

**Theorem 7.7.** Let $\Sigma$ be an (extended) order sorted signature, let $\mathcal{A}, \mathcal{A}'$ be $\Sigma$-algebras, and let $\phi_{\mathcal{A}', \mathcal{A}} : s_0^{\mathcal{A}'} \rightarrow s_0^{\mathcal{A}}$ be a domain homomorphism with regard to domain mapping $id_{TG} : \mathcal{T}G_{\Sigma} \rightarrow \mathcal{T}G_{\Sigma}$, the identity function on $\mathcal{T}G_{\Sigma}$, with $s_0$ the prescribed result type of $\mathcal{T}G_{\Sigma}$-terms. If $\mathcal{A}'$ is an algebra such that each value of $\mathcal{U}^{\mathcal{A}'}$ corresponds to at least one ground term, a property which is in the literature often called “no junk” [6], then $\phi_{\mathcal{A}', \mathcal{A}} : s_0^{\mathcal{A}'} \rightarrow s_0^{\mathcal{A}}$ is a homomorphism from $\mathcal{A}'$ to $\mathcal{A}$.

**Proof:** we have to show that:

$$\phi_{\mathcal{A}', \mathcal{A}}(op^{\mathcal{A}'}(v_1, ..., v_n)) = op^{\mathcal{A}}(\phi_{\mathcal{A}', \mathcal{A}}(v_1), ..., \phi_{\mathcal{A}', \mathcal{A}}(v_n)),$$

for all operators $op$ and all possible value sequences $v_1, ..., v_n$ (values which fulfill the type restriction, definition 2.28). Well, the assumptions imply that there are terms $t_1, ..., t_n$ such that $Val_{\mathcal{A}'}(t_1) = v_1, ..., Val_{\mathcal{A}'}(t_n) = v_n$, so:
7.5. REINTERPRETING OPERATIONS

\[ \text{op}^A(v_1, ..., v_n) = \text{(definition } Val \text{ and assumptions)} \]

\[ Val_A(op(t_1, ..., t_n)), \]

and therefore:

\[ \phi_{A',A}(\text{op}^A(v_1, ..., v_n)) = \phi_{A',A}(Val_A(op(t_1, ..., t_n))). \]

Now:

\[ \phi_{A',A}(Val_A(op(t_1, ..., t_n))) = \text{(definition domain homomorphism)} \]

\[ Val_A(\phi(op(t_1, ..., t_n))) = (\phi = id) \]

\[ Val_A(op(t_1, ..., t_n)) = \text{(definition } Val \text{)} \]

\[ op^A(Val_A(t_1), ..., Val_A(t_n)) = (\phi = id) \]

\[ op^A(Val_A(\phi(t_1)), ..., Val_A(\phi(t_n))) = \text{(definition domain homomorphism)} \]

\[ op^A(\phi_{A',A}(Val_A(t_1)), ..., \phi_{A',A}(Val_A(t_n))) = \text{(assumed)} \]

\[ op^A(\phi_{A',A}(v_1), ..., \phi_{A',A}(v_n)), \]

which we had to prove. Note that the “no junk” requirement implies that \( \phi_{A',A} \) has type \( U^{A'} \rightarrow U^A \).

\[ \square \]

**Theorem 7.8.** Let \( \Sigma, \Sigma', \Sigma'' \) be (extended) order sorted signatures, let \( A, A', A'' \) be \( \Sigma \), respectively \( \Sigma' \), respectively \( \Sigma'' \)-algebras, and let \( \phi_2: TG_{\Sigma''} \rightarrow TG_{\Sigma'} \) and

\[ \phi_1: TG_{\Sigma'} \rightarrow TG_{\Sigma} \]

be surjective domain mappings. Let \( \phi_{A',A''}: s_0^{A''} \rightarrow s_0^{A'} \) and

\[ \phi_{A',A}: s_0^{A'} \rightarrow s_0^{A} \]

be domain homomorphisms with respect to \( A' \), \( A'' \) and \( A \) respectively. The \( \phi(\rho) \) is a domain homomorphism with respect to \( \Sigma, \Sigma' \), and \( \Sigma'' \).

\[ \phi_{A',A} \circ \phi_{A'',A'}: s_0^{A''} \rightarrow s_0^{A} \]

is a domain homomorphism with respect to \( \rho \).

**Proof:** straightforward.

\[ \square \]

**Corollary 7.9.** Let \( \Sigma, \Sigma' \) be (extended) order sorted signatures, let \( A \) be a \( \Sigma \)-algebra, and let \( A', A'' \) be \( \Sigma \)-algebras. Let \( \phi: TG_{\Sigma'} \rightarrow TG_{\Sigma} \) be a surjective domain mapping, \( \phi_{A',A}: s_0^{A'} \rightarrow s_0^{A} \) a domain homomorphism with respect to \( \phi \), and \( \phi_{A''} : s_0^{A''} \rightarrow s_0^{A'} \) a homomorphism from \( A'' \) to \( A' \), with \( s_0 \) and \( s_0' \) the prescribed result types of respectively \( \Sigma \) and \( \Sigma' \). Now \( \phi_{A',A} \circ \phi_{A''} : s_0^{A''} \rightarrow s_0^{A} \) is a domain homomorphism with respect to \( \rho \).

**Proof:** a direct consequence of theorem 7.8 and 7.10: the homomorphism \( \phi_{A''} : s_0^{A''} \rightarrow s_0^{A} \) is a domain homomorphism with respect to \( id_{TG_{\Sigma'}} \). Composition now leads to the desired result: \( \phi \circ id_{TG_{\Sigma'}} = \phi \)

\[ \square \]
In the remaining part of this section for each of the elementary transformations a description will be given of how to reinterpret types and operators in order to get valid interpretations for the transformed signature. This reinterpretation can be seen as a "transformation of models". A model will be derived which is domain homomorphic with the original model, and an appropriate domain homomorphism will be given. Corollary 7.9 leaves us a lot of freedom to derive domain homomorphic algebras; each domain homomorphic algebra comes with an infinite set of homomorphic variants. We will derive algebras which are as close as possible to the original algebra. With six of the elementary transformations it is possible to derive domain homomorphic algebras with the identity function as domain homomorphism (peculiar enough this is not the set of six identity transformations). With the remaining two it is necessary to resort to non-identity, even non-bijective, value mappings.

For each case an informal correctness reasoning will be given. In the schemes for deriving new domains and new operations from original ones, a double prime will (as usual) be used to indicate the new domain. Some derivations can be rendered best in a decomposed "two step" manner. In that case a single prime will be used to indicate the intermediate result.

In the schemes for root side identification and root position identification the binary operator $\overbrace{\cdot}^{\cdot}$ (over-brace) constructs new values from two given values. The only property it has to have is $\overbrace{v_1v_2}^{\cdot} = \overbrace{v_3v_4}^{\cdot} \Leftrightarrow v_1 = v_3 \land v_2 = v_4$ for all values $v_1, v_2, v_3, v_4$.

### 7.5.2 Leaf Side Distinguishing

#### Reinterpretation of Domains

The set of types $S$ was extended with new types which were constructed with the $\overline{\cdot}$ and $\overline{\overline{\cdot}}$-operators. The domains for these new types will be indicated with the same overbar and double overbar notation.

- $\overline{t^A} = t^A$, for all $t \leq s_p$,
- $\overline{\overline{t^A}} = t^A$, for all $t \leq s_p$,
- $\overline{s^A} = s^A$, for all $s \in S$.

#### Reinterpretation of Operations

- $\overline{f^A} = f^A$,
- $\overline{f_2^A} = f^A$,
- $\overline{g^A} = g^A$, for all $g \neq f$.

#### Domain Homomorphism
7.5. REINTERPRETING OPERATIONS

\[ \text{id}_{s_0^A} : s_0^A \to s_0^A \]

Explanation

The set inclusion requirement of definition 2.28 is not strict, so \( s_1 < s_2 \) does not preclude \( s_1^A = s_2^A \). It is easy to see that the newly introduced types fulfill this requirement. The reinterpretation of operations is simple in this case: \( f^A \) will usually be defined for a larger domain than \( f_1^A \) and \( f_2^A \) need to be. This is no problem however: the only requirement is that the operations \( f_1^A \) and \( f_2^A \) are defined rightly for their restricted domain.

7.5.3 Root Side Distinguishing

Reinterpretation of Domains

The set of types \( S \) was extended with new types which were constructed with the \( \top \) and \( \bot \)-operators. The domains for these new types will be indicated with the same overbar and double overbar notation.

- \( \top^n^A = t^A \), for all \( t \) with \( s_r \leq t \),
- \( \bot^n^A = t^A \), for all \( t \) with \( s_r \leq t \),
- \( s^n^A = s^A \), for all \( s \in S \).

Reinterpretation of Operations

- \( f_1^n^A = f^A \),
- \( f_2^n^A = f^A \),
- \( g^n^A = g^A \), for all \( g \neq f \).

Domain Homomorphism

\[ \text{id}_{s_0^A} : s_0^A \to s_0^A \]

Explanation

The domains of the new types are defined to be equal with their origin. This is possible because the definition requires no proper inclusion for the domains if the types are related, i.e. \( s_1 < s_2 \) does not preclude \( s_1^A = s_2^A \). Like with leaf side distinguishing the reinterpretation of operations is simple: same remarks apply.
7.5.4 Root Position Distinguishing

Reinterpretation of Domains

The reinterpretation for root position distinguishing is essentially a repeated application of the scheme for root side distinguishing. The set of types $S$ was extended with new types which were constructed with the $\overline{\tau}$ and $\overline{\tau}$-operators. The domains for these new types will be indicated with the same overbar and double overbar notation.

- \( t''^A = t^A \), for all \( 1 \leq i \leq n \) and all \( t \) with \( s_{ri} \leq t \),
- \( \overline{t''}^A = t^A \), for all \( 1 \leq i \leq n \) and all \( t \) with \( s_{ri} \leq t \),
- \( s''^A = s^A \), for all \( s \in S \).

Reinterpretation of Operations

- \( f''^A = f^A \), for all \( 1 \leq i \leq n \),
- \( f''^A = f^A \), for all \( 1 \leq i \leq n \),
- \( g''^A = g^A \), for all \( g \not\in \{ f_i \mid 1 \leq i \leq n \} \).

Domain Homomorphism

\[ id_{s''^A} : s''^A \to s''^A \]

Explanation

The domains of the new types are defined to be equal with their origin. This is possible because the definition requires no proper inclusion for the domains if the types are related, i.e. \( s_1 < s_2 \) and \( s_1^A = s_2^A \) is permitted. Like with leaf side distinguishing the reinterpretation of operations is simple: same remarks apply.

7.5.5 Leaf Side Identification

Reinterpretation of Domains

The set of types $S$ was extended with new types which were constructed with the $\overline{\tau}$ and $\overline{\tau}$-operators. The domains for these new types will be indicated with the same overbar and double overbar notation. Moreover, the operators $\overline{\tau}$ and $\overline{\tau}$ will also be used to construct new domain values from given values.

- \( t''^A = \{ \overline{v} \mid v \in t^A \} \), for all \( t \leq s_p \),
- \( \tau''^A = \{ \overline{v} \mid v \in t^A \} \), for all \( t \leq s_p \),
7.5. REINTERPRETING OPERATIONS

- \( s^nA = \text{if } s < s_p \)
  
  then \( s^A \cup \{ \overline{v}, \overline{v} \mid v \in s^A \} \)

  \( \text{elif } s_p \leq s \)
   
  then \( s^A \cup \{ \overline{v}, \overline{v} \mid v \in s^A_p \} \)

  \( \text{else } s^A. \)

Reinterpretation of Operations

- \( f'^A = \lambda x_1...x_n. \text{if } x_p \in \overline{s^A} \text{ then } f_1^A(c(x_1), ..., c(x_n)) \text{ else } f_2^A(c(x_1), ..., c(x_n)), \)

- \( g'^A = \lambda x_1...x_n.g^A(c(x_1), ..., c(x_n)), \) for all \( g \not\in \{f_1, f_2\}, \)
  
  with \( c : \mathcal{U}^A \cup \overline{s^A} \cup \overline{s^A_p} \rightarrow \mathcal{U}^A \) the conversion operation defined by:

  \( c(x) = \text{if } x = \overline{v} \in \overline{s^A} \text{ then } v \text{ else } x, \)

- \( g'^A = \text{if } g \in O_{ch,1} \)

  then \( c_1 \circ g'^A \)

  \( \text{elif } g \in O_{ch,2}, \)
   
  then \( c_2 \circ g'^A \)

  \( \text{else } g'^A, \)

  for all \( g \in O_2, \) with \( c_1 \) and \( c_2 \) the conversion operations defined by:

  - \( c_1 = \{ \overline{v} \leftarrow v \mid v \in s^A_p \}, \)
  
  - \( c_2 = \{ \overline{v} \leftarrow v \mid v \in s^A \}. \)

Domain Homomorphism

\( \{ v \leftarrow c(v) \mid v \in s'^A \}, \)

with \( c : \mathcal{U}^A \cup \overline{s^A} \cup \overline{s^A_p} \rightarrow \mathcal{U}^A \) the earlier defined conversion operation.

Explanation

To be able to discriminate between values meant for \( f_1^A \) and values meant for \( f_2^A \) two shadow type structures have been introduced. As opposed to the situation with leaf side distinguishing these types have to correspond to domains which are disjoint with the original domains (it has to be decided based upon the identity of these values whether \( f_1^A \) or \( f_2^A \) should be applied). The new distinct values are constructed with the \( \overline{\tau} \) and \( \overline{\overline{\tau}} \)-operators (which is a natural form of overloading).
CHAPTER 7. SEMANTIC ASPECTS OF TRANSFORMATIONS

The other domains have to be extended with some (or all) $\tau$-values or $\overline{\tau}$-values to satisfy the subtype-subset requirement (definition 2.1). The $c$-operation converts $\tau$-values and $\overline{\tau}$-values to their original counterparts and the $c_1$- and $c_2$-operations convert original values to their $\tau$-value or $\overline{\tau}$-value.

The domain homomorphism cannot be $id$ in this case because operations with $\tau$-converted or $\overline{\tau}$-converted values can occur at root positions. It is easy to see that $c$ is and cannot be not bijective: for each $v \in s^A_\nu$, $v, \overline{v}, \overline{\overline{v}}$ are mapped to the same value: $v$.

7.5.6 Root Side Identification

Reinterpretation of Domains

- $ss^A = \{e_1e_2 | e_1 \in s^A_1, e_2 \in s^A_2\}$,
- $t^A = t^A \cup ss^A$, for all $s_{1r} \leq t$,
- $\overline{t}^A = t^A \cup ss^A$, for all $s_{2r} \leq t$.

Reinterpretation of Operations

- $f^A = \lambda x_1...x_n.\overline{f_1^A(x_1,...,x_n)f_2^A(x_1,...,x_n)}$,
- $g^A = g^A$, for all $g \not\in \{f_1, f_2\}$,
- $g^\nu = \lambda x_1...x_n.g^A(\text{if } \langle g, 1 \rangle \in O_{pa,1} \text{ then } c_1(x_1) \text{ else } x_1)$
  
  
  (\text{if } \langle g, 2 \rangle \in O_{pa,2} \text{ then } c_2(x_1) \text{ else } x_1,)

  
  \vdots

  
  \text{if } \langle g, n \rangle \in O_{pa,1} \text{ then } c_1(x_n)

  
  \text{elif } \langle g, n \rangle \in O_{pa,2} \text{ then } c_2(x_n) \text{ else } x_n).

with $c_1$ and $c_2$ the conversion operators defined by:

$c_1 = \lambda x.\text{if } x = \overline{x_1x_2} \in ss^A \text{ then } x_1 \text{ else } x$,

$c_2 = \lambda x.\text{if } x = \overline{x_1x_2} \in ss^A \text{ then } x_2 \text{ else } x$.

Domain Homomorphism

$id^A_\nu : s^A_\nu \to s^A_0$
7.5. REINTERPRETING OPERATIONS

Explanation

The operators in $O_{pa}$ have to take the $f_1$-value (first component) or the $f_2$-value (second component) of the result of $f$, based on membership of either $O_{pa,1}$ or $O_{pa,2}$. Besides that, operators in $O_{pa}$ have to keep their ability to handle “non-composed” values from other operators. That is why the result of $f^A$ is composed and why the operators in $O_{pa}$ need preprocessing for their operands. If the operand value for one of the operators from $O_{pa}$ is a composed value, then the right component has to be selected from this value. Otherwise the value can be passed on to the operation immediately.

The domain homomorphism can be id because operators to be distinguished (an therefore operations with composed result values) are not allowed at root positions of terms.

7.5.7 Root Position Identification

Reinterpretation of Domains

The reinterpretation for root position identification is essentially a repeated application of the scheme for root side identification.

- $ss_i^{A} = \{ (e_1, e_2) | e_1 \in s_{1ri}^A, e_2 \in s_{2ri}^A \}$, for all $1 \leq i \leq n$,
- $\tilde{t}^A = t^A \cup ss_i^A$, for all $1 \leq i \leq n$ and all $s_{1ri} < t$,
- $\overline{t_i}^A = t^A \cup ss_i^A$, for all $1 \leq i \leq n$ and all $s_{2ri} < t$.

Reinterpretation of Operations

- $f_i^A = \lambda x_1 \ldots x_n. \overline{f_i^A(x_1, \ldots, x_n)} f_{2i}^A(x_1, \ldots, x_n)$, for all $1 \leq i \leq n$,
- $g^A = g^A$, for all $g \notin \{ f_{1i}, f_{2i} | 1 \leq i \leq n \}$,
- $g_i^A = \lambda x_1 \ldots x_n. g^A(c_2(x_1), \ldots, c_2(x_n))$,

for all $g$, with $c_2$ the selection operation defined by:

$c_2 = \lambda x. \text{if } \exists 1 \leq i \leq n \ [x = \overline{x_1x_2} \in ss_i^A] \ then \ x_2 \ else \ x$.

Domain Homomorphism

$\lambda x. \text{if } \exists 1 \leq i \leq n \ [x = \overline{x_1x_2} \in ss_i^A] \ then \ x_1 \ else \ x$.

Explanation

The situation is similar to the root side identification case, except that a set of pairs of operators are being identified in this case. Only the selection operation $c_2$ is present in the expression for $g_i^A$ because it is known that the $x_1$-value is only needed at the top (root-level) of the term. The $c_1$ selection operation can in fact be found in the domain homomorphism.
7.5.8 Contraction

Reinterpretation of Domains

First alternative:

- $t''^A = t^A$, for all $t \leq s_p$,
- $t''^A = t^A$, for all $t \in S$.

Second alternative:

- $t''^A = t^A$, for all $s_p \leq t$,
- $t''^A = t^A$, for all $t \in S$.

Reinterpretation of Operations

- $e''^A = \lambda x_1...x_{m+n-1}.pa^A(x_1,...,x_{p-1},ch^A(x_p,...,x_{p+n-1}),x_{p+n},...,x_{m+n-1})$.
- $g''^A = g^A$, for all $g \not\in \{pa, ch\}$.

Domain Homomorphism

$id''^{s_0}_A : s_0^A \rightarrow s_0^A$

Explanation

The $\tau$-domains and $\tau'$-domains can simply be taken to be equal to the original domains with both alternative approaches because the subtype-subset requirement is non-strict. The reinterpretation of operations is in essence a functional composition of the $pa^A$ and $ch^A$ operations.

7.5.9 Expansion

Reinterpretation of Domains

- $ss''^A$: a new domain for which there are no requirements with regard to containment in other domains because the $ss$-type is not related with other types; there is a requirement for the number of elements in $ss''^A$, but it depends on the interpretation of the new operators $pa^A$ and $ch^A$, to be defined hereafter.

Reinterpretation of Operations

- $pa''^A$ and $ch''^A$ have to be such that for its composition:

  $\lambda x_1...x_m.pa''^A(x_1,...,x_{p-1},ch''^A(x_p,...,x_{p+n-1}),x_{p+n},...,x_m) = c^A$,

- $g''^A = g^A$, for all $g \neq c$. 
Domain Homomorphism

\[ id_{s^A} : s_0^A \rightarrow s_0^A \]

Explanation

The expansion case is easy. Two operations have to be found which have a composition that equals \( c^A \). If \( ch''^A \) is taken to be such that each combination of input values for \( ch''^A \) is mapped to a distinct value of \( ss^A \), then it is always possible to define a \( pa''^A \) that fulfills the requirements. However, it is often possible to find a solution with a smaller set of intermediate values \( ss^A \). A lower-bound for the number of values in \( ss^A \) is:

\[
\max \{ \left| \{ c^A(v_1, \ldots, v_m) / v_p \in t_p^A, \ldots, v_{p+n-1} \in t_{p+n-1}^A \} \right| / \]

\[
v_1 \in t_1^A, \ldots, v_{p-1} \in t_{p-1}^A, v_p \in t_p^A, \ldots, v_m \in t_m^A \}\).

To avoid confusion, the vertical bar of the set comprehension notation is printed as a slash in this expression.

7.5.10 Simplifying Type Systems

With regard to algebraic semantics the scheme for identification of two types has to be completed with a derivation of the domain which has to be associated with the new types. The following can be said about the situation in which types can be identified. If \( \text{Fits}(s_1) = \text{Fits}(s_2) \) then either \( s_1 < s_2 \) or \( s_2 < s_1 \) (provided that \( s_1 \neq s_2 \) and \( \text{Fits}(s_1) \neq \emptyset \neq \text{Fits}(s_2) \)).

It is easy to see that taking:

\[ s^A = \text{if } s_1 < s_2 \text{ then } s_2^A \text{ else } s_1^A, \]

always results in a system in which type domains are consistent with all operations (if \( \text{Fits}(s_1) = \text{Fits}(s_2) = \emptyset \) then any choice for a domain for \( s \) is consistent with the operations), however there are situations in which a narrower domain is possible. Suppose \( \text{Fits}(s_1) \neq \emptyset \neq \text{Fits}(s_2) \), implying \( s_1 < s_2 \) or \( s_2 < s_1 \). If the largest of \( s_1, s_2 \) does not occur as result type of any operator then the domain of \( s \) can be taken to be the domain of the least of \( s_1, s_2 \), since no operand position will be confronted with a value outside this domain. The other situation is the situation in which there are no occurrences of the least of \( s_1, s_2 \) as operand types of operators (otherwise \( \text{Fits}(s_1) \) and \( \text{Fits}(s_2) \) would not have been equal). In this case it is impossible to take the domain of \( s \) to be any narrower than the domain of the largest of \( s_1, s_2 \) without further information of the operators.
Chapter 8

The Transformation Space

8.1 Introduction

In this chapter an exploration will be presented of the space which is generated by the elementary transformations of chapter 6 and the composition primitive. Each of the eight introduced elementary transformations was shown to correspond to a simple ground step simulation, i.e. the following relation was proven to hold between the involved vocabularies $\Sigma$ and $\Sigma'$, bijective domain mapping $\phi : \mathcal{T}G_{\Sigma} \rightarrow \mathcal{T}G_{\Sigma'}$ and rewrite systems $RS$ and $RS'$:

$$\forall t_1, t_2 \in \mathcal{T}G_{\Sigma}[t_1' \rightarrow_{RS'} t_2' \Leftrightarrow \phi(t_1') \rightarrow_{RS} \phi(t_2')]$$

The question arises if it is possible to support any bijective domain mapping between any two sets of ground terms over vocabularies with a sequence of transformations that reduces a rewrite system for one term domain into a rewrite system for the other term domain. The answer is simple: this is not possible. The term sets $\mathcal{T}G_{\Sigma}$ and $\mathcal{T}G_{\Sigma'}$ are in general countably infinite. The number of different bijective functions between two countably infinite sets is non-countable (Cantor's diagonal). However, the number of different transformations that can be composed, based on eight elementary transformations is at most countably infinite, so there are not enough transformations.

In order to explore the transformation space a number of sophisticated transformations will be defined, which can be decomposed into elementary transformations. The sophisticated transformations to be presented represent classes of transformations which are subclasses of the class which is generated by the eight elementary transformations. The transformations to be presented show what kind of transformations can be composed with the eight elementary transformations. They do not cover the entire space stretched by the elementary transformations. However, the transformations to be presented represent interesting subclasses and they do illustrate the potential of the eight elementary transformations, which seem rather weak when viewed on their own.
The situation can be illustrated graphically as follows. The two classes of transformations to be described in this chapter are the restructuring transformations and the generalized identity transformations.

![Diagram](image)

*Figure 8.1: the transformation space*

The rest of this chapter is organized as follows. In section 8.2 a formal apparatus for the description of the transformations is developed. In section 8.3 a sophisticated transformation based (mainly) on elementary contractions and expansions, the restructuring transformation, will be presented. In section 8.4 a number of sophisticated transformations based on elementary distinguishings, the generalized identity transformations, will be presented.

### 8.2 Basic Definitions

**Definition 8.1.** Let $A = (C, V, S, \alpha, \rho, \mu)$ be a vocabulary, $T_A$ the set of terms over $A$ and $FT_A$ the set of fragments over $A$. The set of decompositions $DT_A$ of terms over vocabulary $A$ is defined inductively as the smallest set for which:

- $(fr, c) \in DT_A$,
  - if $fr \in FT_A$ and $\text{Rank}(fr) = 0$,
- $(fr, [dc_1, ..., dc_n]) \in DT_A$,
  - if $fr \in (FT_A - \{\Box\})$, $\text{Rank}(fr) = n$, and $dc_1, ..., dc_n \in DT_A$.

The set $DTG_A$ of ground term decompositions is defined as:

$$DTG_A = \{ dc \in DT_A \mid \text{Comp}(dc) \in TG_A \}.$$
8.2. BASIC DEFINITIONS

The sets $\mathcal{DR}, \mathcal{DRG}, \mathcal{DT}, \mathcal{DTG}$ are defined similarly as subsets of $\mathcal{DT}$. The set of decompositions $\mathcal{DFT}_A$ of fragments over vocabulary $A$ is defined inductively as the smallest set for which:

- $\langle \square, \varepsilon \rangle \in \mathcal{DFT}_A,$

- $\langle fr, \varepsilon \rangle \in \mathcal{DFT}_A,$
  
  if $fr \in \mathcal{FT}_A$ and $Rank(fr) = 0,$

- $\langle fr, [dc_1, ..., dc_n] \rangle \in \mathcal{DT}_A,$
  
  if $fr \in (\mathcal{FT}_A - \{\square\}),$ $Rank(fr) = n,$ and $dc_1, ..., dc_n \in \mathcal{DFT}_A.$

It is easy to see that $\mathcal{DT}_A \subseteq \mathcal{DFT}_A,$ like $\mathcal{T}_A \subseteq \mathcal{FT}_A$. The set $\mathcal{DFTG}_A$ of ground fragment decompositions is defined as:

$$\mathcal{DFTG}_A = \{ dc \in \mathcal{DFT}_A \mid Comp(dc) \in \mathcal{FTG}_A \}.$$ 

The sets $\mathcal{DFR}, \mathcal{DFRG}, \mathcal{DFT}, \mathcal{DFTG}$, are defined similarly as subsets of $\mathcal{DFT}$. We do not need each individual set of decompositions. Most of these sets are defined for completeness.

The function $Comp : \mathcal{DFT}_A \to \mathcal{FT}_A,$ that associates each decomposition with one unique fragment is defined recursively as:

- $Comp(\langle \square, \varepsilon \rangle) = \square,$

- $Comp(\langle fr, \varepsilon \rangle) = fr,$
  
  if $fr \in \mathcal{FT}_A$ and $Rank(fr) = 0,$

- $Comp(\langle fr, [dc_1, ..., dc_n] \rangle) = fr[Comp(dc_1), ..., Comp(dc_n)],$
  
  if $fr \in (\mathcal{FT}_A - \{\square\}),$ $Rank(fr) = n,$ and $dc_1, ..., dc_n \in \mathcal{DFT}_A.$

It is said that $dc \in \mathcal{DFT}_A$ is a decomposition of $fr \in \mathcal{FT}_A$ if and only if $Comp(dc) = fr.$

\begin{flushright}
$\square$
\end{flushright}

\textbf{Definition 8.2.} Let $A$ be a vocabulary. The \textit{Root} operator will be overloaded in a natural way for root-extraction of decompositions. $\text{Root} : \mathcal{DFT}_A \to \mathcal{FT}_A.$ Formal definition:

$$\text{Root}(\langle fr, [dc_1, ..., dc_n] \rangle) = fr,$$ for all $fr \in \mathcal{FT}.$

The \textit{Root} operator (definition 5.4) is overloaded and extended to decompositions. Note that $\mathcal{DFT}_A \cap \mathcal{FT}_A = \emptyset$ for all $A$, a necessary requirement for the resolution of overloadedness.

\begin{flushright}
$\square$
\end{flushright}

\textbf{Definition 8.3.} Let $A$ be a vocabulary. The set of positions in a decomposition, denoted by $\text{Pos} : \mathcal{DFT} \to \mathcal{N}^*,$ is defined formally as:
\[ \text{Pos}(\emptyset, \varepsilon) = \{\varepsilon\}, \]
\[ \text{Pos}(\langle fr, \varepsilon \rangle) = \{\varepsilon\}, \]
for all \( fr \in FA \),
\[ \text{Pos}(\langle fr, [dc_1, \ldots, dc_n] \rangle) = \{i.\pi \mid 1 \leq i \leq n, \pi \in \text{Pos}(dc_i)\}, \]
for all \( fr \in FA \) and all \( dc_1, \ldots, dc_n \in DFA \).

The operator \( \text{Pos} \) (definition 5.22) is overloaded and extended to decompositions. Each decomposition \( dc \) of a term \( t = \text{Comp}(dc) \) defines a relation between positions in \( \text{Pos}(dc) \) and positions in \( \text{Pos}(t) \). Informally: a position \( p_{dc} \in \text{Pos}(dc) \) is related to a position \( p_t \in \text{Pos}(t) \) if and only if \( p_t \) is a position in the fragment with position \( p_{dc} \) in decomposition \( dc \). A formal definition of the relation \( R_{dc} \) which is determined by decomposition \( dc \) is as follows:

\[ R_{dc} \subseteq \text{Pos}(dc) \times \text{Pos}(\text{Comp}(dc)) \]

with:

- \( \varepsilon R_{dc} s \iff s \in \text{Pos}(\text{Root}(dc)) \),
- \( i.s R_{dc} s_1.s_2 \iff fr|s_1 = \emptyset \land s_2 R_{dc} s, \text{ when } dc = \langle fr, [dc_1, \ldots, dc_n] \rangle. \)

\[ \square \]

**Definition 8.4.** Let \( A \) be a vocabulary. The **context decomposition at position** \( p \in \text{Pos}(dc) \) in a decomposition \( dc \in DFA \), denoted by \( dc|_p \), is defined formally as:

- \( dc|_\varepsilon = dc \),
- \( \langle fr, [dc_1, \ldots, dc_n] \rangle|_{i.p} = dc_i|_p. \)

The operator \( \text{".|"} \) (definition 5.23) is overloaded and extended to decompositions.

\[ \square \]

**Definition 8.5.** Let \( A \) be a vocabulary. The **replacement in** \( dc \in DFA \) of a decomposition at position \( p \in \text{Pos}(dc) \) by a decomposition \( dc_{new} \in DFA \), denoted by \( dc[dc_{new}]_p \), is defined formally as:

- \( dc[dc_{new}]_\varepsilon = dc_{new} \),
- \( \langle fr, [dc_1, \ldots, dc_n] \rangle[dc_{new}]_{i.p} = \langle fr, [dc_1, \ldots, dc_i[dc_{new}]_p, \ldots, dc_n] \rangle. \)

The operator \( \text{"[.]"} \) (definition 5.24) is overloaded and extended to decompositions.

\[ \square \]

**Definition 8.6.** Let \( A \) be a vocabulary. The function \( \text{Ffrags} : DFA \rightarrow \mathcal{P}(FA) \) returns the set of fragments that occur in a given decomposition. A recursive definition:
8.2. BASIC DEFINITIONS

- \( \text{Frags}(\Box, \epsilon) = \emptyset \),
- \( \text{Frags}(\{e\}) = \{\{e\}\} \), for all \( \text{Rank} - 0 \) fragments \( fr \in FT_A \),
- \( \text{Frags}(\{fr, \lfloor dc_1, \ldots, dc_n \rfloor\}) = \{\{fr\}\} \cup \bigcup_{i=1}^{n} \text{Frags}(dc_i) \), for all \( \text{Rank} - n \) fragments \( fr \in FT_A \).

\[\square\]

**Definition 8.7.** Let \( A \) be a vocabulary. A fragment \( fr' \in FT_A \) is called a subfragment of a fragment \( fr \in FT_A \) if and only if \( fr' \neq \Box \) and there is a position \( p \in Pos(fr) \) such that \( fr \mid_p = fr' \). If \( p \neq \epsilon \) then \( fr \) may also be called a proper subfragment of \( fr \).

\[\square\]

**Definition 8.8.** Let \( A \) be a vocabulary. Two fragments \( fr, fr' \in \mathcal{FRG}_A \) are called unifiable if and only if there are fragments \( fr_1, \ldots, fr_n, fr'_1, \ldots, fr'_n \in \mathcal{FRG}_A \) such that \( fr[fr_1, \ldots, fr_n] = fr'[fr'_1, \ldots, fr'_n] \).

\[\square\]

**Definition 8.9.** Let \( A \) be a vocabulary. Two fragments \( fr, fr' \in \mathcal{FRG}_A \) are said to be overlapping each other if and only if:

\[
\text{neither of them equals } \Box \text{ and } \\
( \text{there is a subfragment of } fr \text{ which is unifiable with } fr' \text{ or } \\
\text{there is a subfragment of } fr' \text{ which is unifiable with } fr )
\]

The parentheses indicate the higher priority of or over and. If \( fr \) overlaps \( fr \) then \( fr \) will be said to be self overlapping.

\[\square\]

**Definition 8.10.** Let \( A \) be a vocabulary. A fragment \( fr \in \mathcal{FRG}_A \) is called a generalization of a fragment \( fr' \in \mathcal{FRG}_A \) if and only if there are fragments \( fr_1, \ldots, fr_n \in \mathcal{FRG}_A \) such that \( fr' = fr[fr_1, \ldots, fr_n] \). If at least one of \( fr_1, \ldots, fr_n \neq \Box \) then \( fr \) may also be called a proper generalization.

\[\square\]

**Definition 8.11.** Let \( A \) be a vocabulary. A fragment \( fr \in \mathcal{FRG}_A \) is said to be contained in a fragment \( fr' \in \mathcal{FRG}_A \) if and only if \( fr \neq \Box \) and there is a position \( p' \in Pos(fr') \) such that \( fr \) is a generalization of \( fr'|_{p'} \).

\[\square\]

**Definition 8.12.** Let \( A \) be a vocabulary and \( dc \in DFT_A \) a decomposition of a term \( t = \text{Comp}(dc) \). A fragment \( fr \) which is contained in \( t \) is said to be covered by a decomposition \( dc \) if and only if all non-hole positions in the fragment (these are the positions of the fragment in the term augmented with all non-hole positions in the
fragment) are related to the position of one particular fragment in the decomposition, in the relation between positions that was defined in definition 8.3.

\textbf{Theorem 8.13.} Let $A$ be a vocabulary and $fr, fr' \in \mathcal{FRG}_A$. If fragment $fr'$ is contained in fragment $fr$ then $fr$ and $fr'$ overlap each other. \textit{Proof:} easy.

\textbf{Definition 8.14.} Let $A$ be a vocabulary and $fr, fr' \in \mathcal{FRG}_A$. The fragments $fr, fr'$ are said to be \textit{properly overlapping} each other if and only if:

- neither of them equals $\Box$ and
- ( there is a subfragment of $fr$ which is unifiable with $fr'$ and of which $fr'$ is not a generalization) or
- ( there is a subfragment of $fr'$ which is unifiable with $fr$ and of which $fr$ is not a generalization

The parentheses indicate the priorities of the various occurrences of the \textit{and} and \textit{or} operators. Note that two fragments one of which is contained in the other can also be properly overlapping. Take for instance the fragments:

$$f(g(h(g(\Box)))),$$

and:

$$g(h(\Box)).$$

For that reason it is not sufficient to define "proper overlap" as "overlap and no containment".

\textbf{Definition 8.15.} Let $A$ be a vocabulary. A subset $B_A \subseteq \mathcal{FRG}_A$ is called a \textit{term base} for term set $\mathcal{T}_A$ if and only if:

1. $\forall t \in \mathcal{T}_A \exists dc \in D\mathcal{T}_A[Comp(dc) = t \land Frags(dc) \subseteq B_A],$
2. $\forall fr, fr' \in B_A [fr$ and $fr'$ are not properly overlapping].$

Note that the second requirement does not preclude the possibility of $fr$ and $fr'$ being equal. This means that self overlapping fragments like $c(c(\Box))$ can never be members of bases.

\textbf{Example 8.16.} Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ be a vocabulary. The following two subsets of $\mathcal{FRG}_A$ can easily be shown to be bases for $\mathcal{T}_A$:
8.2. BASIC DEFINITIONS

1. \( \{c(\square, \ldots, \square) \mid c \in C\} \),

2. \( TG_A \).

The first basis if finite if and only if \( A \) is finite, which was assumed for any vocabulary in this document (definition 5.1). The second basis is in general not finite.

\[ \]

Definition 8.17. A decomposition over a basis \( B_A \subseteq FRG_A \) of a term \( t \in TG_A \) is a decomposition \( dc \in DTG_A \) for which \( Comp(dc) = t \) and \( Frags(dc) \subseteq B_A \). The coarsest decomposition over a basis \( B_A \subseteq FRG_A \) of a term \( t \in TG_A \) is a decomposition \( dc \) of \( t \) for which \( \mid Frags(dc) \mid \) is minimal.

\[ \]

Lemma 8.18. Let \( A \) be a vocabulary and \( B_A \) a basis for \( A \). If two basis fragments from \( B_A \) are overlapping then one of them is contained in the other.

\[ \]

Proof: easy.

\[ \]

Theorem 8.19. Let \( B_A \subseteq FRG_A \) be a basis for \( TG_A \). Each term \( t \in TG_A \) has one uniquely determined coarsest decomposition.

\[ \]

Proof: Let \( dc \) and \( dc' \) be two decompositions with equal, minimal numbers of fragments. We will show that \( dc = dc' \) with induction on the number of fragments in \( dc \). Let \( dc = \langle fr, [dc_1, \ldots, dc_n] \rangle \), \( dc' = \langle fr', [dc'_1, \ldots, dc'_n] \rangle \). If \( fr = fr' \) then the induction hypothesis can be applied to the pairs \( dc_i, dc'_i \) for each \( 1 \leq i \leq n = n' \), and we are done. Suppose \( fr \neq fr' \). Definition 8.15 implies that either \( fr \) is contained in \( fr' \) or \( fr' \) is contained in \( fr \). Suppose, without loss of generality, that \( fr' \) is contained in \( fr \). This means that \( n' > 0 \). Definition 8.15 implies also that at least one of the fragments in \( \{Root(dc' \mid i) \mid 1 \leq i \leq n' \} \) is contained in \( fr \) and that there are positions \( p_1, \ldots, p_n \in Pos(dc) \) (lexicographically ordered) such that \( Comp(dc' \mid p_i) = Comp(dc_i) \) for all \( 1 \leq i \leq n \) (lemma 8.18). It is easy to see now that:

\[ dc[dc' \mid p_1] \ldots [dc' \mid p_n] \]

must have less fragments than \( dc' \), contradicting the assumptions. Hence \( fr = fr' \) and \( dc = dc' \).

\[ \]

Theorem 8.20. Let \( B_A \subseteq FRG_A \) be a basis for \( TG_A \). The coarsest decomposition over \( B_A \) of a term \( t \in TG_A \) can be determined in a top-down fashion without the need for backtracking, i.e. given a term \( t \in TG_A \) the coarsest decomposition over \( B_A \) can be determined by searching for the largest pattern \( s \in B_A \) that matches root-to-root with \( t \), and by recursively proceeding in the same manner with the sub-terms of \( t \) which correspond to the holes in the found pattern in \( B_A \) (the size of a fragment is simply the number of non-hole positions in it, \text{Numhp}, definition 5.36).

\[ \]

Proof: the statement follows immediately from the fact that if the fragments of a basis overlap then one is always contained in the other (lemma 8.18).
Definition 8.21. Let $A$ be a vocabulary and $B_A$ a term base for $A$. A fragment $fr \in \mathcal{FRG}_A$ is called present in $B_A$ if and only if:

- it is explicitly present in $B_A$ or
- if it is implicitly present in $B_A$,

explicit and implicit presence of fragments being defined as:

- A fragment $fr \in \mathcal{FRG}_A$ is explicitly present in $B_A$ if and only if $fr \in B_A$.
- A fragment $fr \in \mathcal{FRG}_A$ is implicitly present in $B_A$ if and only if in each term $t \in \mathcal{T}_A$ in which this fragment is contained, this fragment is covered by a different fragment of the base in a coarsest decomposition.

Example 8.22. Consider the vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ with:

$$
C = \{f, g, h, a\},
$$

$$
\alpha(\{f, g, h\}) = \{1\},
$$

$$
\rho(f, 1) = \{g, h\},
$$

and consider the base $B_A$ for $\mathcal{T}_A$:

$$
B_A = \{f(g(\square)), f(h(\square)), f(\square), g(\square), h(\square), a\}.
$$

The fragment $f(\square)$ is implicitly present in $B_A$ because below $f$ there is either $g$ or $h$, so all occurrences of $f(\square)$ in terms from $\mathcal{T}_A$ are covered by either $f(g(\square))$ or by $f(h(\square))$. The fragment $f(\square)$ is also explicitly present in $B_A$. It is an easy to establish fact that fragments which are explicitly as well as implicitly present in a base, can be removed from that base without invalidating the requirements for term bases.

Definition 8.23. Let $A$ be a vocabulary and $B_A$ a term base for $\mathcal{T}_A$. Let $fr \in \mathcal{FRG}_A$ be a fragment which is present in $B_A$. Now $fr$ is called complete in $B_A$ if and only if:

- $fr$ consists of a single constructor, or:
- there is a decomposition of $fr$ into two fragments, both constituting fragments of which are present in $B_A$ and complete in $B_A$. 

8.3. RESTRUCTURING TRANSFORMATIONS

The entire base $B_A$ is called complete if and only if all the fragments which are explicitly present in $B_A$ are complete in $B_A$.

\[ \square \]

**Theorem 8.24.** Let $A$ be a vocabulary and $B_A$ a finite, complete term base for $\mathcal{T}G_A$. There exists a sequence of contractions which contracts in each term each occurrence of a base fragment in that term into one single constructor (a one constructor fragment).

**Proof.** A constructive proof will be given. A process for the derivation of such a sequence from a given base will be described.

Contract each fragment that consists of two or more constructors as follows. This is an iteration construction over the set of fragments. A description for the contraction of individual fragments will be given.

According to the definition of completeness of bases, each fragment consisting of two or more fragments can be decomposed in at least one way into two subfragments which are present and complete in $B_A$. If one, or both, of these fragments consist of two or more constructors: contract these fragments by recursively applying the prescription for the contraction of individual fragments. Afterwards: contract the two resulting single constructor fragments into one single constructor.

Fragments which are implicitly present in the base will never occur in decompositions of terms, so any base fragment occurrence will be contracted into one single constructor by the contraction sequence derived above.

\[ \square \]

8.3 Restructuring Transformations

8.3.1 Introduction

In this section a category of transformations will be presented with which it is possible to change the structure of terms. Each of these transformations can be decomposed into a sequence of contractions, expansions and identifications. The contractions and expansions form the essential components, identifications are necessary because combinations of contractions and expansions sometimes include implicit distinguishing, which can be reversed by identifications.

8.3.2 The Reversal of Contractions

The elementary transformations of chapter six have been presented as pairs of counterparts, i.e. leaf side distinguishing is the counterpart of leaf side identification, root side distinguishing the counterpart of root side identification, etc. A complication with contraction and expansion is that they are not true counterparts; there are situations in which a contraction transformation cannot be reversed by an expansion (though an expansion can always be reversed by a contraction). In some circumstances there are additional identifications necessary to "undo" the result of a contraction transformation. An example will be given to illustrate this.
Example 8.25. Contraction reversed by expansion and identification. Consider the vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ with:

$$C = \{f, g, a\},$$
$$S = \{f, g\},$$
$$\alpha(\{f, g\}) = \{1\}, \alpha(a) = 0,$$
$$\rho(f, 1) = \{g, a\}, \rho(g, 1) = \{f, a\},$$
$$V, \mu: \text{not relevant}.$$ 

It is easy to see that $\mathcal{T}\mathcal{G}_A$ contains the $f$-starting terms:

$$f(a), f(g(a)), f(g(f(a))), f(g(f(g(a))), \ldots$$

and the $g$-starting terms:

$$g(a), g(f(a)), g(f(g(a))), g(f(g(f(a))), \ldots$$

Contracting $f$ and $g$ (at position 1 of $f$) into $h$ leads to a modified vocabulary $A'$ with:

$$C' = \{f, g, h, a\},$$
$$S' = \{f, g, h\},$$
$$\alpha'(\{f, g, h\}) = \{1\}, \alpha'(a) = 0,$$
$$\rho'(f, 1) = \{a\},$$
$$\rho'(g, 1) = \{f, h, a\},$$
$$\rho'(h, 1) = \{f, h, a\}.$$ 

The set $\mathcal{T}\mathcal{G}_{A'}$ contains:

$$f(a), h(a), h(f(a)), h(h(a)), \ldots$$

and:

$$g(a), g(f(a)), g(h(a)), g(h(f(a))), \ldots$$

Expanding $h$ into $f'$ and $g'$ leads to vocabulary $A''$ with:

$$C'' = \{f, f', g, g', a\},$$
$$S'' = \{f, f', g\},$$
\[ \alpha''(\{f, f', g, g'\}) = \{1\}, \alpha''(a) = 0, \]
\[ \rho''(f, 1) = \{a\}, \]
\[ \rho''(f', 1) = \{g'\}, \]
\[ \rho''(g, 1) = \{f, f', a\}, \]
\[ \rho''(g', 1) = \{f, f', a\}. \]

and the set \( \mathcal{TG}_{A''} \) contains:
\[ f(a), f'(g'(a)), f'(g'(f(a))), f'(g'(f'(g'(a)))), \ldots \]

and:
\[ g(a), g(f(a)), g(f'(g'(a))), g(f'(g'(f(a)))), \ldots \]

which is clearly different from \( \mathcal{TG}_A \). The original vocabulary can be restored by performing a leaf side identification on \( f, f' \) and performing a root position identification on \( g, g' \). It is easy to verify that the conditions for these identifications hold (section 6.7 and 6.8). Note that the two identifications will result in two new constructors distinct from \( f \) and \( g \) (constructors which do not already occur in the original vocabulary), but we consider two vocabularies which are identical up to systematic renaming of constructors to be equal.

It is easy to see that a composed contraction and expansion transformation, like in this example, will in general require two additional identifications to restore the original situation: the contraction introduces one new constructor and the expansion replaces this constructor by two new constructors. Two identifications are necessary to identify these constructors with their origins. The number of necessary identifications is reduced if the contraction transformation makes one or two constructors superfluous because they are non-integrated or unsatisfiable. □

**Theorem 8.26.** Each contraction transformation can be reversed by one expansion transformation, followed by zero, one or two identification transformations, and each expansion transformation can be reversed by one contraction transformation.

**Proof:** Easy. □

**Theorem 8.27.** Each sequence of contraction transformations can be reversed by a sequence of expansion transformations, followed by a sequence of identification transformations.

**Proof:** By repeated application of theorem 8.26: each sequence of contractions can be reversed by a sequence of pairs of expansions and identifications. An example with three contractions (schematically):

\[ contr_1, contr_2, contr_3, exp_3, id_3, exp_2, id_2, exp_1, id_1. \]
Now it is possible to postpone the identifications until after the last expansion, yielding:

\[\text{contr}_1, \text{contr}_2, \text{contr}_3, \exp_3, \exp_2, \exp_1, \id_3, \id_2, \id_1.\]

The legitimacy of this reordering of transformations follows from the fact that the expansions which are moved from after an identification to before this identification do not interfere with that identification. This is easy to see because all expansions operate on distinct new constructors (new because they came into existence in the preceding contraction sequence) and their result consists also of entirely new constructors. Non-interference follows now from the non-overlapping of involved patterns and standard theory about confluence. □

### 8.3.3 Formal Definition of Restructuring Transformations

It appears that it is possible to succeed a sequence of contractions by a sequence of expansions and identifications which do not lead to the original vocabulary but to a vocabulary in which some or all terms have a structure which is different from the structure of their originals. For instance: terms which contain the fragment \(f(g(\Box, \Box), h(\Box))\) can be transformed into terms that contain the fragment \(f'(g'(h'(\Box, \Box, \Box)))\), which has a fundamentally different structure. A formal definition of this class of transformations and their corresponding domain mappings is as follows.

**Definition 8.28.** Let \(A\) and \(A'\) be vocabularies and \(\phi : \mathcal{T}G_A \rightarrow \mathcal{T}G_{A'}\) a total, bijective domain mapping. Now \(\phi\) is called a **restructuring domain mapping** if and only if there is a term base \(B_A \subseteq \mathcal{FR}G_A\) for \(\mathcal{T}G_A\), a term base \(B_{A'} \subseteq \mathcal{FR}G_{A'}\) for \(\mathcal{T}G_{A'}\), and there are functions:

- \(\phi_D : \mathcal{D}TG_A \rightarrow \mathcal{D}TG_{A'}\),
- \(\psi : B_A \rightarrow B_{A'}\),

such that \(\psi\) is bijective and for all terms \(t \in \mathcal{T}G_A\) with coarsest decomposition \(dc = (fr, [dc_1, ..., dc_n])\) over \(B_A\):

- \(\text{Comp}(\phi_D(dc)) = \phi(\text{Comp}(dc))\),
- \(\phi_D((fr, [dc_1, ..., dc_n])) = (\psi(fr), [\phi_D(dc_1), ..., \phi_D(dc_n)])\),
- \(\phi_D(dc)\) is a coarsest decomposition of \(\phi(t)\) over \(B_{A'}\).

The function \(\psi\) is called the **characteristic function** of \(\phi\). The only purpose of the function \(\phi_D\) is to fix the relation between \(\phi\) and \(\psi\): \(\phi_D\) is \(\phi\) on decomposed terms. If we want to associate the particular base for which there exist functions \(\phi_D\) and \(\psi\) such that \(\psi\) describes \(\phi\) via \(\phi_D\), we say that \(\phi\) is **restructuring with regard**
to base $B_A$. Note that with $\mathcal{TG}_A$, $\phi$, $B_A$, the functions $\phi_D$ and $\psi$ are uniquely determined. A restructuring transformation will be defined to be a transformation which corresponds to a restructuring domain mapping.

Example 8.29. An example will be presented of a restructuring domain mapping and its characteristic function $\psi$. Consider the vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

\[
C = \{f_1, f_2, g_1, g_2, g_3, g_4, a\},
\]
\[
\alpha(\{f_1, f_2\}) = \{2\}, \alpha(\{g_1, g_2, g_3\}) = \{1\}, \alpha(a) = 0,
\]
\[
\rho(f_1, 1) = \rho(f_1, 2) = \{g_1, g_2, g_3\},
\]
\[
\rho(f_2, 1) = \rho(f_2, 2) = \{g_1, g_2, g_3, g_4\},
\]
\[
\rho(g_1, 1) = \ldots = \rho(g_4, 1) = \{f_1, f_2, a\},
\]
\[
S = \{f_1, f_2, a\},
\]
\[
V, \mu: \text{not relevant},
\]

and vocabulary $A' = \langle C', V', S', \alpha', \rho', \mu' \rangle$ with:

\[
C' = \{f_1', g_1', g_2', g_3', g_4', g_5', a\},
\]
\[
\alpha'(f_1') = 2, \alpha'(\{g_1', g_2', g_3', g_4', g_5'\}) = \{1\}, \alpha'(a') = 0,
\]
\[
\rho'(f_1', 1) = \rho'(f_1', 2) = \{g_1', g_2', g_3', g_4', g_5'\},
\]
\[
\rho'(g_1', 1) = \ldots = \rho'(g_5', 1) = \{f_1', a'\},
\]
\[
S' = \{f_1', a'\},
\]
\[
V', \mu': \text{not relevant},
\]

Terms in $\mathcal{TG}_A$ consist of fragments $f_i(g_{j_1}(\square), g_{k_1}(\square))$ with $i \in \{1, 2\}$, $j_1, k_1 \in \{1, 2, 3\}$, $j_2, k_2 \in \{1, 2, 3, 4\}$ and the fragment $a$. It is easy to see that there are $9 + 16 = 25$ fragments with the shape $f_i(g_{j_1}(\square), g_{k_1}(\square))$, namely $f_1$ with $3 \times 3$ combinations of $g_x$, and $f_2$ with $4 \times 4$ combinations of $g_x$. Terms in $\mathcal{TG}_{A'}$ consist of fragments $f'_i(g_{j_1}(\square), g_{k_1}(\square))$, namely $f_1$ with $5 \times 5$ combinations of $g_x$. A bijective domain mapping can now be constructed by defining a correspondence between $a$ and $a'$ and a bijective function from the 25 fragments of $\mathcal{TG}_A$ to the 25 fragments of $\mathcal{TG}_{A'}$:

- $\psi(a) = a'$,
- $\psi(f_1(g_1(\square), g_1(\square))) = f'_1(g'_1(\square), g'_1(\square))$,
- $\psi(f_1(g_1(\square), g_2(\square))) = f'_1(g'_1(\square), g'_2(\square))$, 
-

\[ \psi(f_1(g_1(\square), g_2(\square))) = f'_1(g'_1(\square), g'_2(\square)), \]
\[ \psi(f_1(g_2(\square), g_1(\square))) = f'_1(g'_1(\square), g'_4(\square)), \]
\[ \vdots \]
\[ \psi(f_2(g_4(\square), g_4(\square))) = f'_1(g'_4(\square), g'_2(\square)). \]

The non-triviality of the restructuring transformation notion lies in that it is often not possible to decompose elements of bases into smaller fragments. \( \square \)

**Theorem 8.30.** A restructuring transformation can be decomposed in a sequence of elementary contractions, followed by a sequence of elementary expansions, followed by a sequence of elementary identifications.

**Proof:** The truth of the statement is an easy consequence of the previous theorems. According to theorem 8.24 both \( A \) and \( A' \) can be transformed into vocabularies (say \( B \) and \( B' \)) in which each constructor corresponds one-to-one with a fragment in the coarsest decomposition of the original term. The characteristic function \( \psi \) of the restructuring transformation corresponds to a bijection between the constructors of \( B \) and \( B' \). So \( B \) and \( B' \) are identical up to renaming, which is identical from our point of view. The proof is completed by stating that the contraction sequence which contracted \( A' \) into \( B' \) can be reversed in a sequence of expansions, followed by a sequence of identifications that transform \( B' \) into \( A' \) (theorems 8.26 and 8.27). \( \square \)

**Theorem 8.31.** The class of restructuring transformations (class \( C \) in figure 8.1) is not closed under composition.

**Proof:** Consider a vocabulary \( A \) with unary constructors \( f \) and \( g \) and consider the composition of the two very simple restructuring transformations consisting of the contraction of \( g \) below \( f \), succeeded by the contraction of \( f \) below \( g \). Bases for the description of these transformations as restructuring transformations contain respectively the fragments \( f(g(\square)) \) and \( g(f(\square)) \). It is easy to see that there is no single restructuring transformation with the same net effect as the composition of the two contractions. The fragments \( f(g(\square)) \) and \( g(f(\square)) \) cannot be member of one base because they are properly overlapping. Any composition of \( f(g(\square)) \) and \( g(f(\square)) \) (notably \( f(g(f(\square))) \) or \( g(f(g(\square))) \)) is also not allowed because of proper self overlap. \( \square \)

The following can be said about the relation between the classes \( A, B, C, \) and \( D \) from figure 8.1. Class \( B \) is properly contained in class \( A \) (an argument for this has been presented in section 8.1). Class \( C \) and \( D \) are not disjunct, but their intersection contains only trivial transformations. The union of class \( C \) and class \( D \) is properly contained in class \( B \). Any composition of a contraction and a distinguishing transformation is in \( B \) but not in \( C \) nor in \( D \).
8.3.4 Justification for Design Decisions

A rationale for the design of the term base notion and the restructuring transformation concept is as follows. The prohibition of proper overlap among base fragments (second requirement of definition 8.15) is because properly overlapping base fragments will introduce ambiguity in the determination of decompositions. It is possible to resolve this ambiguity by imposing an ordering on the base fragments. A possible solution is to impose a total ordering on base fragments by placing them in a sequence instead of a set. The imposition of an ordering on the fragments leads to more complicated derivation processes of contraction sequences and to more complicated theorems about decompositions. The imposition of orderings on term bases will therefore be left as a subject for further research.

The necessity of the completeness requirement for term bases of restructuring transformations will be illustrated with an example. Consider a vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

$$C = \{f, g, h, a\},$$
$$\alpha(\{f, g, h\}) = \{1\},$$
$$\rho(f, 1) = \rho(g, 1) = \rho(h, 1) = C,$$

and consider the base:

$$B_A = \{f(g(h(\Box))), f(\Box), g(\Box), h(\Box), a\}.$$

It is impossible to contract occurrences of the fragment $f(g(h(\Box)))$ with our elementary contraction transformation without affecting the rest of the term set in some other way. The contraction of $f(g(h(\Box)))$ has to start with either the contraction of $f(g(\Box))$ or the contraction of $g(h(\Box))$, which will have its impact on independent occurrences of the fragments $f(g(\Box))$ or $g(h(\Box))$. Expansion of the remaining results of the first contraction after the second contraction is not a real solution because the new constructors cannot be identified with their original counterparts. Consider for instance the following transformation sequence:

1. contraction of $f(g(\Box))$ into $f_{g}(\Box)$,
2. contraction of $f_{g}(h(\Box))$ into $f_{g}h(\Box)$,
3. expansion of $f_{g}(\Box)$ into $f'(g'(\Box))$.

This transformation sequence will result in a vocabulary in which the conditions for the identification of $f$, $f'$ and of $g$, $g'$ do not hold, as can be verified easily.

A different approach would be to perform a leaf side distinguishing transformation on $g$ and to call $g$ with $h$ at its operand position $g'$. Contracting $f(g'(\Box))$ now will not affect independent occurrences of $f(g(\Box))$, but it will affect independent occurrences of $g(h(\Box))$ since after the $f(g'(\Box))$ contraction $g$ and $g'$ can no longer
be identified into the one constructor of the original situation. The message is that it is just impossible to contract a fragment consisting of three or more constructors into a one constructor fragment, with multiple elementary contractions, and without affecting the rest of the term, or other terms, in some way. The independent contraction of fragments with three or more constructors is not impossible because of inherent reasons but because the elementary two-constructor contraction is not powerful enough. A new basic transformation has to be defined for that. To speak in terms of figure 8.1: an independent contraction transformation for a fragment consisting of three or more constructors is in space \( A \) but not in space \( B \).

The development of a generalized contraction transformation for the independent contraction of fragments consisting of three or more constructors is left as a subject for further research.

### 8.4 Generalized Identity Transformations

In this section a number of sophisticated identity transformations will be presented. The first three ones are just generalizations of certain identity transformations, the last one will be shown to be decomposable into elementary identity transformations.

**Definition 8.32.** Let \( A = (C, V, S, \alpha, \rho, \mu) \) be a vocabulary with \( c \in C \). The transformation **generalized leaf side distinguishing** position \( p \) of constructor \( c \) with partition \( \{s_1, ..., s_m\} \) of \( \rho(c, p) \) is defined as the composition of a number of distinguishing transformations. It is defined as leaf side distinguishing \( c \) into \( c_1 \) and \( c'_2 \) with partition \( \{s_1, s_2 \cup ... \cup s_m\} \), followed by leaf side distinguishing \( c'_2 \) into \( c_2 \) and \( c'_3 \) with partition \( \{s_2, s_3 \cup ... \cup s_m\} \), ..., followed by leaf side distinguishing \( c'_{m-1} \) into \( c_{m-1} \) and \( c_m \) with partition \( \{s_{m-1}, s_m\} \). The transformation **generalized root side distinguishing** is defined similarly.

It is easy to see that the generalized distinguishing transformations are well-defined. Choosing a sequence different from \( s_1, ..., s_m \) will lead to the same results, or at least results which are identical up to systematic renaming of constructors.

**Definition 8.33.** Let \( A = (C, V, S, \alpha, \rho, \mu) \) be a vocabulary with \( c \in C \). The transformation **complete leaf side distinguishing** operand position \( p \) of constructor \( c \) is defined as generalized leaf side distinguishing operand position \( p \) of constructor \( c \) with partition \( \{\{c_1\}, ..., \{c_n\}\} \) of \( \{c_1, ..., c_n\} = \rho(c, p) \).

The transformation **complete root side distinguishing** constructor \( c \) is defined as a root position distinguishing of \( c \) into \( c_r \) and \( c_l \), followed by a generalized root side distinguishing of \( c_r \) with partition \( \{\{(c_1, p_1)\}, ..., \{(c_n, p_n)\}\} = \rho^{-1}(c_r) \).

**Definition 8.34.** The transformation **complete distinguishing** constructor \( c \) is defined to be a composition of complete leaf side distinguishing \( c \) with regard to each operand position of \( c \) **together with** complete root side distinguishing \( c \).

After a complete distinguishing transformation on \( c \), occurrences of represen-
tatives of $c$ in the new vocabulary reflect completely the immediate context of $c$: the identity of the representatives tell us exactly which constructors occur above or below $c$, and whether or not $c$ is a root constructor.

\[\square\]

**Definition 8.35.** The transformation *complete distinguishing a vocabulary* $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ is defined to be the least transformation such that afterwards each constructor reflects the identity of each adjacent constructor in the original situation. The determination “least” means that no more distinctness between constructors is allowed than strictly necessary for this definition.

The “least” requirement means that if complete distinguishing a vocabulary is to be attained by consecutive distinguishings of individual constructors, then only the first constructor to be distinguished needs to be distinguished completely. From the subsequent constructors to be distinguished, the distinguishings do not have to be completely if previously distinguished constructors occur in $\rho$- or in $\rho^{-1}$-values. For instance, if $\rho(c_1, 1) = \{c_2, c'_2, c''_2\}$ and $c_2$ has been distinguished into $\{c_{21}, \ldots, c_{2n}\}$, then $\rho(c_1, 1) = \{c_{21}, \ldots, c_{2n}, c'_2, c''_2\}$ in the new situation. Now when distinguishing $c_1$ to reflect the identity of the constructors below it in the original situation, it is sufficient of perform leaf side distinguishing, based on partition $\{\{c_{21}, \ldots, c_{2n}\}, \{c'_2\}, \{c''_2\}\}$, since each of the $c_{2t}$ represents the original $c_2$.

\[\square\]
Chapter 9

Transformation Strategies

9.1 Introduction

The elementary transformations, as introduced in chapter 6, can be used to simplify rewrite systems. Simplifications or complexity reductions have a price. The transformation of a rewrite system involves two kinds of calculations: manipulations with terms to be rewritten and manipulations with sets of rules. For each of the elementary transformations as presented in chapter 6 it is easy to show that the average asymptotic time and space complexity of the procedural descriptions of the manipulations with rules is linear in reasonable input measures such as number of rules and total number of constructors in patterns of rules. Also the manipulation with terms to be rewritten (defined by rewrite systems in chapter 6) can easily be seen to be implementable with a time complexity which is at most linear in the size of terms.

Any cost of a transformation of a set of rules can be justified by a large enough number of terms to be rewritten. The cost of transforming rewrite rules is one-off, so there is always an area at the right of a break-even point in which transformation is cost-effective. Another cost of complexity reduction by transformation is the cost of both domain transformations. We have chosen to neglect these costs for a number of reasons.

The first reason is that the cost of building a tree(term) representation is usually indifferent for small modifications in the term structure. Terms to be rewritten are usually not given as such but have to be constructed, for instance in a parsing process. The construction of a transformed term to be rewritten can be accomplished by a modification of the series of actions for the construction of the tree (term) representation during parsing. This modification will in general have little or no impact on the cost of the parsing process. Similar arguments hold for the back transformation. Un-parse or print processes have a cost which is to a large extent indifferent for particular ways of representation.

The second reason is that the manipulations in the new domain may comprise more than the application of just one transformed rewrite system. It may include storage and retrieval of the tree structured data in databases, comparisons of this
tree structured data and other operations which can be done regardless of the way in which specific data is represented.

Thirdly: if the cost cannot be eliminated or reduced in the two ways just mentioned, the domain transformations of the elementary transformations of chapter 6 are all very simple transformations which are cheap in terms of CPU and memory usage.

There are at least four quantities associated with vocabularies and rewrite systems which can be subjected to optimization:

1 the number of rules in the rewrite system,

2 the number of constructor occurrences in well-formed terms and patterns of rules,

3 the number of different constructors in the constructor set,

4 particular properties such as integratedness, satisfiability, being trivial combination free, being normalized, being uniquely starting (definition 5.9 through 5.13) and also properties like modular decomposability and properties with regard to the arity of constructors.

The properties mentioned in the fourth item are discrete quantities but it is very well possible to regard transformations that establish these properties as optimizations or complexity reductions.

All four of the quantities just mentioned are more or less contradictory in the sense that transformations that optimize one or two of them often do so at the expense of one or two other quantities. Transformations are nevertheless useful because improvements in one direction may outweigh deteriorations in another direction. This may be because of circumstances in which improvements in one direction are valued higher than deteriorations in another direction. The usefulness of certain transformations and transformation strategies depends therefore heavily on contextual factors.

It will be explored how to combine the various elementary transformations of chapter 6 in order to achieve a certain simplification goal. The strategies that will be presented to optimize the various criteria are mostly "greedy" or just slightly more sophisticated than a simple greedy approach. The question whether it is possible to reach an optimum in all situations for each of the criteria with the presented strategies is not addressed in this chapter. It is left as an interesting subject for further research.

The structure of this chapter parallels the presentation of the work hypothesis in chapter 1. For each item in the work hypothesis there is an associated section in this chapter. Section 9.2 is a general section which presents some useful new transformations for the rest of the chapter. Section 9.3 deals with transformation strategies to reduce the number of rewrite rules, section 9.4 with transformation strategies to reduce the sizes of terms, section 9.5 with transformation strategies to reduce the number of different operators involved in systems, and section 9.6 finally
9.2. IDENTICAL DOMAIN TRANSFORMATIONS

deals with strategies to attain various desirable properties, like a many sorted type system and the establishment of modular decomposability.

9.2 Identical Domain Transformations

9.2.1 Introduction

In this section four categories of transformations will be introduced that affect only the rule set of a system, i.e. both term domains are identical and more importantly: the domain mapping defined on this domain is the identity function. In section 9.2.2 and 9.2.3 two generalizing transformations and in section 9.2.4 and 9.2.5 two specializing transformations will be presented. The principle of generalizing transformations is to replace a set of rules with one rule, which is more general. The principle of specializing transformations is to replace one rule with a set of rules, each of which is more specialized, more specific, than the original rule. Specializations are the inverse transformations of generalizations. Specializations are not very interesting transformations as such because a specialization increases both pattern sizes and numbers of rules. However, specializations may enable other transformations. One important merit of specializations is that collapsing rules (definition 5.26) are replaced by sets of non-collapsing rules. Specializations can therefore be utilized to eliminate collapsing rules from a rewrite system. A specific advantage of leaf side specializations, is that they can eliminate non-maximally typed variables (definition 5.20).

With the identical domain transformations to be introduced, the transformed rewrite systems behave exactly the same as the original system. Another way of saying this is that the transformed rewrite systems are simple step simulators (definition 3.5) of the original system with the identity function as domain mapping. The consequence of the fact that both domains are identical and that the domain mapping equals the identity function, is that the transformations need no description of how to derive a simulating vocabulary and no definition of a domain mapping. The transformations will be presented in the form of schemes which describe when and how to replace a subset of rules with one rule or how to replace one rule with a set of rules.

A thorough formal approach requires a formal proof of simple step simulation for each of the generalizing and each of the specializing transformations. We will not give these proofs. The principles behind generalization and specialization are not difficult and each of the transformations can easily intuitively be seen to be correct. Instead, we will give some informal remarks about the necessity of the various conditions that have to hold for a transformation to be applicable.

9.2.2 Leaf Side Generalization

Leaf side generalization is a generalization in which a set of rules is replaced by one single, more general rule, based on the elimination of the constructors which
occur directly above variables of the patterns (which play the role of leaves, hence the name) at some specific positions of the patterns.

**Scheme 9.1.** A general scheme for the leaf side generalization transformation for a vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \). The rule subset consisting of \( m \) rules:

\[
\{ fr[c_1(v_{c_1,1}, \ldots, v_{c_1,\alpha(c_1)}), \ldots, c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)})) \rightarrow fr'[c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)}), \ldots, c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)})) | 1 \leq i \leq m, v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)} are, for each i, distinct variables which are all maximally typed and which do not occur in fr nor in fr' \},
\]

can be replaced by the single rule:

\[
fr[v, \ldots, v] \rightarrow fr'[v, \ldots, v],
\]

if the following condition is satisfied:

1. \( v \) is a new variable which does not already occur in \( fr \) and with \( \mu(v) = \{ c_i | 1 \leq i \leq m \} \).

The fragment notation \( fr[...] \) was introduced in definition 5.34 and 5.37.

**Explanation.** The variables \( v_{c_{i,j}} \) have to be distinct and maximally typed because multiple occurrences of one variable, as well as non-maximally-typedness, may impose an additional condition on the firing of a rule, which cannot be incorporated in the “reduced” rule \( fr[v, \ldots, v] \rightarrow fr'[v, \ldots, v] \). The variables \( v_{c_{i,j}} \) are not allowed to occur in \( fr \) nor \( fr' \), i.e. all multiple occurrences of \( v_{c_{i,j}} \) have to be covered by substitutes \( c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)}) \), because occurrences of one or more of the \( v_{c_{i,j}} \) in \( fr \) or \( fr' \) would impose an additional condition on the firing of a rule which cannot be incorporated in the “reduced” rule \( fr[v, \ldots, v] \rightarrow fr'[v, \ldots, v] \).

The additional condition, \( v \) a new variable which does not already occur in \( fr \) and with \( \mu(v) = \{ c_i | \ldots \} \) is to prevent interference with variables which already occur in \( fr \) and to maintain the functionality of the original rule set. In section 9.3.2 some measures are described to establish the conditions for this generalization if they do not already hold.

Note that if \( \text{Rank}(fr) = 0 \), then necessarily \( \text{Rank}(fr') = 0 \) (implied by the third requirement of definition 5.21) and the transformation reduces to the trivial transformation.

**Example 9.2.** The following system allows leaf side generalization. Consider the vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) with:
9.2. IDENTICAL DOMAIN TRANSFORMATIONS

\[ C \ni \{f, g, h, i\}, \]
\[ V \ni \{v_1, v_2, v_3, v_4, v_5\}, \]
\[ S = C, \]
\[ \alpha(\{f, g, h\}) = \{2\}, \alpha(i) = 1, \]
\[ \rho(f, 1) = \{h, i\}, \rho(f, 2) = \mu(v_3), \]
\[ \rho(h, 1) = \mu(v_1), \rho(h, 2) = \mu(v_2), \]
\[ \rho(i, 1) = \mu(v_4). \]

The rules:
\[ \{ f(h(v_1, v_2), v_3) \rightarrow g(h(v_1, v_2), v_3), f(i(v_4), v_3) \rightarrow g(i(v_4), v_3) \} \]
can be replaced by the rule:
\[ f(v_5, v_3) \rightarrow g(v_5, v_3) \]

with \( \mu(v_5) = \{h, i\}. \)

9.2.3 Root Side Generalization

Root side generalization is a generalization in which a set of rules is replaced by one single, more general rule, based on the elimination of the root constructors of the patterns of these rules.

**Scheme 9.3.** A general scheme for the root side generalization transformation for a vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \). The rule subset consisting of \( m \) rules:

\[ \{ c_i(v_{ci,1}, ..., v_{ci,p_i-1}, t, v_{ci,p_i+1}, ..., v_{ci,\alpha(c_i)}) \rightarrow c_i(v_{ci,1}, ..., v_{ci,p_i-1}, t', v_{ci,p_i+1}, ..., v_{ci,\alpha(c_i)}) \mid 1 \leq i \leq m, \]
\[ 1 \leq p_i \leq \alpha(c_i), \]
\[ v_{ci,1}, ..., v_{ci,p_i-1}, v_{ci,p_i+1}, ..., v_{ci,\alpha(c_i)} \text{ are, for each } i, \text{ distinct variables which are all maximally typed and which do not occur in } t \text{ nor in } t' \}
\]
can be replaced by the rule:
\[ t \rightarrow t', \]
if the following conditions are satisfied:

1. $\rho^{-1}(\text{Root}(t)) = \bigcup_{i=1}^{m} \{(c_i, p_i)\}$,

2. $\text{Root}(t) \notin S$. 

Explanation. The variables $v_{n,j}$ have to be distinct and maximally typed because multiple occurrences of one variable, as well as non-maximally-typedness, may impose an additional condition on the firing of a rule, which cannot be incorporated in the “reduced” rule $t \rightarrow t'$. The first additional condition is to ensure that each possible context of $t$ does occur in the original set of rules. If $\rho^{-1}(\text{Root}(t))$ is larger than $\bigcup(c_i, p_i)$ then replacement of the set of rules by $t \rightarrow t'$ would introduce unwanted functionality. The second additional condition is to prevent terms $t$ to be rewritten to $t'$ by the transformed system; if $\text{Root}(t) \notin S$ the term $t$ is not a well-formed term, implying that it will never be subjected to rewriting. Note that if $t$ is well-formed, it will be rewritten by the rule $t \rightarrow t'$ but not by the original set of rules. In section 9.3.2 some measures are described to establish the conditions for this generalization if they do not already hold.

Example 9.4. The following system allows root side generalization. Consider the vocabulary $A = (C, V, S, \alpha, \rho, \mu)$ with:

$$C \supseteq \{f, g, h, a\},$$

$$V \supseteq \{v\},$$

$$S = \{f, g, a\},$$

$$\alpha(f) = 2, \alpha(\{g, h\}) = \{1\}, \alpha(a) = 0,$$

$$\rho(f, 1) \supseteq \{h, a\}, \rho(f, 2) = \mu(v),$$

$$\rho(g, 1) \supseteq \{h, a\},$$

$$\rho(h, 1) \supseteq \{a\},$$

$$\rho^{-1}(h) = \{(f, 1), (g, 1)\}.$$ 

The rules:

$$\{f(h(a), v) \rightarrow f(a, v), g(h(a)) \rightarrow g(a)\}$$

can be replaced by the rule:

$$h(a) \rightarrow a.$$
9.2. IDENTICAL DOMAIN TRANSFORMATIONS

9.2.4 Leaf Side Specialization

Leaf side specialization is the inverse operation of leaf side generalization. A single rule is replaced by a set of rules, based on the extension of the rule with more specific constructors above some of the variables of the patterns.

**Scheme 9.5.** A general scheme for the leaf side specialization transformation for a vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$. The rule:

$$fr[v, \ldots, v] \rightarrow fr'[v, \ldots, v],$$

satisfying the condition:

$$v \notin \text{Var}(fr), v \notin \text{Var}(fr'),$$

can be replaced by the rule set:

$$\{ fr[c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)}), \ldots, c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)})] \rightarrow$$

$$fr'[c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)}), \ldots, c_i(v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)})] \mid$$

$$1 \leq i \leq |\mu(v)|,$$

$$c_i \in \mu(v),$$

$$v_{c_i,1}, \ldots, v_{c_i,\alpha(c_i)} \text{ are, for each } i, \text{ distinct variables which are all maximally typed and which do not already occur in } fr \text{ nor in } fr'$$

$$\} \alpha.$$ 

The fragment notation $fr[...]$ was introduced in definition 5.34 and 5.37 and the $[\cdot]_\alpha$-operator in definition 5.19. □

**Explanation.** The conditions in the set comprehension are self-evident. The $[\cdot]_\alpha$ operator is to eliminate alpha-variant terms which are generated by the set comprehension because alpha-variant terms are distinct.

The additional condition, $v$ occurs not already in $fr$ nor $fr'$, is to prevent interference with variables which already occur in one or both fragments.

Note that if $\text{Rank}(fr) = 0$, then necessarily $\text{Rank}(fr') = 0$ (implied by the third requirement of definition 5.21) and the transformation reduces to the trivial transformation.

**Example 9.6.** An example that illustrates the ability of leaf side specializations to eliminate collapsing rules. Consider the vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

$$C \ni \{ f, g, h, a \},$$

$$V \ni \{ v_1, v_2, v_3, v_4 \},$$
\[ \alpha(a) = 0, \alpha(\{f, g\}) = \{2\}, \alpha(h) = 1, \]
\[ \rho(f, 1) \ni \{a\}, \rho(f, 2) \ni \mu(v_1), \]
\[ \rho(g, 1) = \mu(v_2), \rho(g, 2) = \mu(v_3), \]
\[ \rho(h, 1) = \mu(v_4), \]
\[ \mu(v_1) = \{g, h\}. \]

The collapsing rule:
\[ f(a, v_1) \rightarrow v_1, \]

can be replaced by the non-collapsing rules:
\[ \{f(a, g(v_2, v_3)) \rightarrow g(v_2, v_3), f(a, h(v_4)) \rightarrow h(v_4)\}. \]

\[ \Box \]

**Example 9.7.** An example that illustrates the ability of leaf side specializations to eliminate variables which are not maximally typed from rules. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

- \( C \supseteq \{a, b, c, f, g, h\} \),
- \( V \supseteq \{v, w\} \),
- \( S = \{a, b, c, h\} \),
- \( \alpha(\{a, b, c\}) = \{0\}, \alpha(\{f, g, h\}) = \{1\} \),
- \( \rho(f, 1) = \{a, b, c, h\}, \rho(g, 1) = \{a, b, c, h\}, \rho(h, 1) = \{a, h\} \),
- \( \mu(\{v, w\}) = \{\{a, h\}\} \).

In the rule:
\[ f(v) \rightarrow g(v) \]

the variable \( v \) is not maximally typed in the left hand side of the rule, because \( \mu(v) \neq \rho(f, 1) \). A leaf side specialization that establishes maximal typedness of the variable \( v \) goes as follows. The original rule has to be replaced by two rules, one in which \( v \) has been replaced by \( a \), and one in which \( v \) has been replaced by \( h(w) \) with \( \mu(w) = \{a, h\} = \rho(h, 1) \). The resulting rule set:
\[ \{f(a) \rightarrow g(a), f(h(w)) \rightarrow g(h(w))\}, \]

can easily be shown to simple-step-simulate the original rule and the only variable in the second rule is maximally typed. \( \Box \)
9.2. IDENTICAL DOMAIN TRANSFORMATIONS

9.2.5 Root Side Specialization

Root side specialization is the inverse operation of root side generalization. A single rule is replaced by a set of rules, based on the extension of the rule with more specific constructors above the root of the patterns.

Scheme 9.8. A general scheme for the root side specialization transformation for a vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \). The rule:

\[ l \rightarrow r, \]

satisfying the condition:

\[ \text{Root}(l) \notin S \]

can be replaced by the rule set:

\[
\{ \{ c_i(v_{c_i,1}, \ldots v_{c_i,p_i-1}, l, v_{c_i,p_i+1}, \ldots v_{c_i,\alpha(c_i)}) \rightarrow c_i(v_{c_i,1}, \ldots v_{c_i,p_i-1}, r, v_{c_i,p_i+1}, \ldots v_{c_i,\alpha(c_i)}) \mid 1 \leq i \leq |\rho^{-1}(\text{Root}(l))|, \langle c_i, p_i \rangle \in \rho^{-1}(\text{Root}(l)), v_{c_i,1}, \ldots, v_{c_i,p_i-1}, v_{c_i,p_i+1}, \ldots, v_{c_i,\alpha(c_i)} \text{ are, for each } i, \text{ distinct variables which are all maximally typed and which do not already occur in } l \} \}_{\alpha}.
\]

The \( \cdot \}_{\alpha} \)-operator was defined in definition 5.19.

Explanation. The conditions in the set comprehension are self-evident. The \( \cdot \}_{\alpha} \)-operator is to eliminate alpha-variant terms which are generated by the set comprehension because alpha-variant terms are distinct.

The additional condition \( \text{Root}(l) \notin S \) is to exclude \( l \) and \( r \) from being rewrite related. If \( l \) (or some instance of \( l \)) rewrites to \( r \) (or some instance of \( r \)) then the original rule \( l \rightarrow r \) has to be maintained because the rewrite relation between \( l \) and \( r \) cannot be defined with the set of scheme 9.8. If \( \text{Root}(l) \notin S \) then \( l \) (and all instances of \( l \)) are not well-formed, so the question whether or not \( l \) and \( r \) are rewrite related is not relevant.

Example 9.9. An example that illustrates the ability of root side specializations to eliminate collapsing rules. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

\[
C \supseteq \{ f, g, h, a \},
\]

\[
V \supseteq \{ v_1, v_2 \},
\]
\[ \alpha(a) = 0, \alpha(\{f, g\}) = 2, \alpha(h) = 1, \]

\[ \rho(f, 1) \ni a, \rho(f, 2) \ni \mu(v_1), \]

\[ \rho^{-1}(f) = \{\langle g, 2 \rangle, \langle h, 1 \rangle\}, \]

\[ \mu(v_2) = \rho(g, 1). \]

The collapsing rule:

\[ f(a, v_1) \rightarrow v_1 \]

can be replaced by the non-collapsing rules:

\[ \{g(v_2, f(a, v_1)) \rightarrow g(v_2, v_1), h(f(a, v_1)) \rightarrow h(v_1)\}. \]

\[ \square \]

9.3 Rule Reductions

9.3.1 Introduction

Strategies for rule reductions, with which we mean a reduction of the number of rules, will be based on the root side and leaf side generalizations that were introduced in section 9.2. Sometimes these generalizations introduce collapsing rules or rules with non-maximally typed variables (section 9.2). If this is not the case, or if collapsing rules and/or non-maximally-typedness poses no problems, then the generalization should be applied immediately. There are virtually no disadvantages associated with directly applying these schemes; the number of rules decreases and also the size of the resulting rule is less than the size of each of the original rules. If the generalization schemes are not directly applicable it is often possible to make the conditions for these schemes to hold with one or more applications of the elementary transformations of chapter 6. However, the elementary syntactic transformations sometimes have as a consequence an increase of the number of rules, so they need to be applied in an intelligent manner.

The various elementary transformations will lead to an increase in the number of rules depending on the shape of the original rules. The transformation “root side identification” for instance will lead to an increase only if the constructor to be distinguished occurs at the root position of some pattern (left or right) of some rule. Below a list will be given of the conditions under which a certain transformation results in an increase in the number of rules, and concrete figures for the amount of increase. The conditions and figures are based on elementary transformations which are “enriched” with transformations (leaf side specializations, section 9.2.5), to keep variables maximally typed in left hand side patterns. Reason for this is that certain transformations are not possible if the variables of the rules are not maximally typed in the left hand sides of rules. Advantages of non-maximal typing of variables with regard to the number of rules can always be taken at the end of a transformation.
sequence. The fact that the operation of one rule has to be simulated by two or more
rules is referred to as rule multiplication in the following list. The figures in the list
can be derived easily from the algorithms in chapter 6. The descriptions in the list
have to be viewed in the context of chapter 6; in that chapter the meaning of the sets
$C_{ch}, C_{pa}$ and $F$ can be found.

- Leaf side distinguishing of $f$ at $p$: rule multiplications only if the constructor to
be distinguished occurs with a variable at the specific distinguishing position $p$
in either or both patterns of the rule or if both root constructors of the patterns
of the rule occur in different subsets of the partition of $C_{ch}$. The number of
extra rules generated by rule $h = \langle l, r \rangle$ is $k \ast |\rho(f, p)|^m \ast 2^m - 1$ with:

- $f$ occurring once or more in left hand side pattern $l$ above $m$ different
variables at position $p$, regardless of whether $f$ occurs above the same
variables in the right hand side pattern $r$,

- $f$ occurring once or more in right hand side pattern $r$ above $n$ different
variables at position $p$ and not in left hand side pattern $l$ above the same
variables at position $p$,

- if $\text{Root}(l) \in C_{ch,i}$ and $\text{Root}(r) \in C_{ch,3-i}$ for some $i \in \{1, 2\}$ then
$k = |\rho^{-1}(\text{Root}(l))|$ else $k = 1$.

- Root side distinguishing of $f$: rule multiplications only if the constructor to be
distinguished occurs as root constructor in either or both patterns of the rule.
The number of extra rules generated by rule $h = \langle l, r \rangle$ is: $k \ast \Pi m_i - 1$ with:

- $k = 2$, if $f$ occurs as root of the left hand side pattern $l$, regardless of
whether $f$ occurs in right hand side pattern $r$,

- $k = |\rho^{-1}(f)|$, if $f$ occurs as root of right hand side pattern $r$ and not as
root of left hand side pattern $l$,

- $k = 1$, if $f$ occurs neither at the root of the left hand side pattern $l$, nor
at the root of the right hand side pattern $r$,

- $n$ the number of variables for which the union of $\text{Par}(v, l)$ and $\text{Par}(v, r)$
is not entirely contained in $C_{pa,1}$ or $C_{pa,2}$ and for each $1 \leq i \leq n$:
$m_i = |\mu(v_i)|$.

- Root position distinguishing of the set of constructors $F$: rule multiplications
only if there are rules with left hand side roots that are member of $F$. For each
distinct rule which has a left hand side root that is member of $F$: 1 extra rule.

- Leaf side identification of $f$ at $p$: rule multiplications only if the constructor to
be distinguished occurs with a variable at the specific identification position $p$
in either or both patterns of the rule. The number of extra rules generated by
rule $h = \langle l, r \rangle$ is $|\rho(f, p)|^m \ast 2^n - 1$ if:
- $f$ occurs once or more in left hand side pattern $l$ above $m$ different variables at position $p$, regardless of whether $f$ occurs above the same variables in the right hand side pattern $r$,
- $f$ occurs once or more in right hand side pattern $r$ above $n$ different variables at position $p$ and not in left hand side pattern $l$ above the same variables at position $p$.

- Root side identification of $f$ at $p$: rule multiplications only if the constructor to be identified occurs as root constructor in either or both patterns of the rule. The number of extra rules generated by rule $h = \langle l, r \rangle$ is:

  - if $f$ occurs as root of the left hand side pattern $l$: $|\rho^{-1}(f)| - 1$ extra rules, regardless of whether $f$ occurs in right hand side pattern $r$,
  - if $f$ occurs as root of right hand side pattern $r$ and not as root of left hand side pattern $l$: 1 extra rule.

- Root position identification of the sets of constructors $F_1$ and $F_2$: no rule multiplications, possibly rule reductions.

- Contraction of $pa$ and $ch$ at position $p$ of $pa$: rule multiplications only if $ch$ occurs at the root position of either or both patterns or if $pa$ occurs with a variable at position $p$ in either or both patterns. The number of extra rules generated by rule $h = \langle l, r \rangle$ is $2^m * 2^n - 1$ with:

  - $m$ the number of different variables $v$ that occur somewhere below $pa$ at $p$ with $\mu(v) \supset \{ch\}$ (note that $\supset \not\Rightarrow \not=$),
  - $n = \text{if } P2(pa, ch, p) \text{ then } 0 \text{ else } 1$ (the predicate $P2$ was introduced in definition 6.1).

- Expansion: no rule multiplications.

In the rest of this section a number of techniques will be presented that enable generalizations or that support the enabling of generalizations. The techniques are based on the various elementary transformations, so the effects of these techniques on the number of rules can be found in the just presented list. The section will be concluded with the presentation of a simple strategy to achieve rule reductions, using the various different techniques.

### 9.3.2 Enabling Generalizations

#### Introduction

If the conditions for the generalizations as described in section 9.2 do not hold, it is sometimes possible to establish these conditions with one or more elementary transformations from chapter 6. In the rest of this subsection it will be discussed
what elementary transformations can be utilized in which situation to make the conditions for the various generalizations to hold. Measures for the two distinct types of generalizations will be discussed separately. The measures will be presented in an informal way as comments on the formal descriptions of the various generalizations. The contexts for these measures are defined in the formal descriptions of the generalizations (section 9.2.2 and 9.2.3).

Leaf Side Generalization

Two simple measures are possible to enable the leaf side generalization on a rule (sub)set:

1. If one of the variables $v_{c_i, n_i} (1 \leq n_i \leq \alpha(c_i))$ is not maximally typed at position $n_i$ of $c_i$ (which is because of the identical contexts in which the $v_{c_i, n_i}$ occur equivalent with $\mu(v_{c_i, n_i}) \subset \rho(c_i, n_i)$, with $\subset \Rightarrow \neq$) then maximal typedness can be obtained by leaf side distinguishing $c_i$ at position $n_i$ with partition $\{\mu(v_{c_i, n_i}), \rho(c_i, n_i) - \mu(v_{c_i, n_i})\}$ of $\rho(c_i, n_i)$.

2. If there are rules in the set of $m$ for which the constructors at the hole positions are not one and the same constructor (there may be differences between the left hand side and the right hand side pattern but also differences between the hole positions of one pattern) then equality can be obtained by root side identification of the two different constructors, provided the conditions for this identification hold (if not: section 9.3.3 might present a solution). Note that leaf side identification is never possible in this situation because of the well-formedness requirements for variables in terms.

**Example 9.10.** An example that illustrates various measures to enable a leaf side generalization. Consider the vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

$C \supset \{f, g, h, i_1, i_2, a, b\}$,

$V \supset \{v_1, v_2, v_3, v_4\}$,

$\alpha(\{f, g, h\}) = \{2\}$, $\alpha(\{i_1, i_2\}) = \{1\}$, $\alpha(\{a, b\}) = \{0\}$,

$\rho(f, 1) = \{h, i_1\}$, $\rho(f, 2) = \mu(v_3)$,

$\rho(g, 1) = \{h, i_2\}$, $\rho(g, 2) \supset \mu(v_3)$,

$\rho(h, 1) = \mu(v_1)$, $\rho(h, 2) = \{f, g, h, i_1, i_2, a, b\}$,

$\rho(i_1, 1) = \mu(v_4)$, $\rho(i_2, 1) = \mu(v_4)$,

$\rho^{-1}(i_1) \cap \rho^{-1}(i_2) = \emptyset$,

$\mu(v_2) = \{f, g, h, i_1, i_2\}$,
and consider the rules:

\[ \{ f(h(v_1, v_2), v_3) \to g(h(v_1, v_2), v_3), f(i_1(v_4), v_3) \to g(i_2(v_4), v_3) \} . \]

A leaf side generalization becomes possible after:

- leaf side distinguishing \( h \) at the second operand position, with partition \( \{ \{ f, g, h, i_1, i_2 \}, \{ a, b \} \} \),
- root side identifying \( i_1 \) and \( i_2 \) (into \( i \) for instance).

The resulting (sub)system is isomorphic (identical up to renaming) with the (sub)system from example 9.4 (\( h_1 \) corresponds to \( h \)) and the two rules can be replaced by the single rule:

\[ f(v_5, v_3) \to g(v_5, v_3) \]

with \( \mu(v_5) = \{ h, i \} \).

\( \square \)

**Root Side Generalization**

Four simple measures are possible to enable the root side generalization on a rule (sub)set.

1. If one of the variables \( v_{c_i n_i} (1 \leq n_i \leq \alpha(c_i)) \) is not maximally typed at position \( n_i \) of \( c_i \) (which is because of the single occurrence of \( v_{c_i n_i} \) the case if and only if \( \mu(v_{c_i n_i}) \subseteq \rho(c_i, n_i) \), with \( \subseteq \Rightarrow \neq \)) then maximal-typedness can be obtained by leaf side distinguishing \( c_i \) at position \( n_i \) with partition \( \{ \mu(v_{c_i n_i}), \lambda(c_i, n_i) - \mu(v_{c_i n_i}) \} \) of \( \rho(c_i, n_i) \).

2. If there are rules in the set of \( m \) rules for which the left hand side root and the right hand side root are not equal then equality can be obtained by leaf side identification of the two different constructors, provided the conditions for this identification hold (if not: section 9.3.3 might present a solution). Note that root side identification is never possible in this situation because of the type-preservingness requirements for rules (section 5.3).

3. If externally specified condition 1. does not hold but \( \rho^{-1}(\text{Root}(t)) \supseteq \bigcup_{i=1}^{m} \{ (c_i, p_i) \} = C_{pa,1} \) (with \( \supseteq \Rightarrow \neq \)) then set equality can be obtained by root side distinguishing \( c = \text{Root}(t) \) with partition \( \{ C_{pa,1}^{-1}(c) - C_{pa,1} \} \) of \( \rho^{-1}(c) \).

4. If externally specified condition 2. does not hold, i.e. \( \text{Root}(t) \in S \) then things can be improved by root position distinguishing \( \text{Root}(t) \). The conditions of the root position discriminating transformation might imply that a set of constructors, including \( \text{Root}(t) \), is subjected to simultaneous root position distinguishing.
Example 9.11. An example that illustrates various measures to enable a root side generalization. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

\[
\begin{align*}
C & \supset \{f, g_1, g_2, h, a, b\}, \\
V & \supset \{v\}, \\
S & \not\ni h, \\
\alpha(f) & = 2, \alpha(\{g_1, g_2, h\}) = \{1\}, \alpha(\{a, b\}) = \{0\}, \\
\rho(f, 1) & \supset \{h, a\}, \rho(f, 2) = \{f, h, a, b\}, \\
\rho(g_1, 1) & \supset \{h, a\}, \\
\rho(g_2, 1) & \supset \{h, a\}, \\
\rho(h, 1) & \supset \{h, a\}, \\
\mu(v) & = \{f, h\}, \\
\rho^{-1}(h) & = \{(f, 1), (g_1, 1), (g_2, 1), (h, 1)\}, \\
\rho^{-1}(g_1) & \cap \rho^{-1}(g_2) = \emptyset,
\end{align*}
\]

and consider the rules:

\[
\{f(h(a), v) \rightarrow f(a, v), g_1(h(a)) \rightarrow g_2(a)\}.
\]

A root side generalization becomes possible after:

- leaf side distinguishing \( f \) at the second operand position, with partition \( \{\{f, h\}, \{a, b\}\} \),

- root side identifying \( g_1 \) and \( g_2 \) (into \( g \) for instance),

- root side distinguishing \( h \) with partition \( \{\{(f, 1), (g, 1)\}, \{(h, 1)\}\} \).

The resulting (sub)system is isomorphic (identical up to renaming) with the (sub) system from example 9.2 \( (f_1 \text{ corresponds to } f \text{ and } h_1 \text{ to } h) \) and can be replaced by the single rule:

\[
h(a) \rightarrow a.
\]
9.3.3 Enabling Identifications

Introduction

If the conditions for the leaf side identification, the root side identification, or the root position identification as described in section 6.7, 6.8, and 6.9 do not hold, it is sometimes possible to establish these conditions with one or more of the other elementary transformations from chapter 6. In the rest of this subsection it will be discussed what elementary transformations can be utilized in which situation to make the conditions for the two mentioned identifications to hold. Measures for the distinct identifications will be discussed separately. The measures will be presented in an informal way as comments on the formal descriptions of the three identifications. The context for these measures can be found in section 6.7, 6.8, and 6.9.

Leaf Side Identification

Five simple measures are possible to enable the leaf side identification transformation. The measures establish the conditions 4 through 8 of the leaf side identification transformation (section 6.7).

1. If the arities of the constructors to be identified are not equal then equality can be attained by increasing the arity of the lowest arity constructor or by decreasing the arity of the highest arity constructor. Both measures can be done with expansion or contraction transformations. Arity modifying transformations are discussed in detail in section 9.6.8 (establishes condition 4).

2. If the constructors which are allowed to occur below $f_1$ and $f_2$ at position $p$ are not disjoint sets then disjointness can be obtained by root side distinguishing the constructors in the overlap between the two sets. If $c \in (\rho(f_1, p) \cap \rho(f_2, p))$ then $c$ should be subjected to root side distinguishing and the partition $\{C_{pa,1}, C_{pa,2}\}$ of $\rho^{-1}(c)$ should be such that $\langle f_1, p \rangle \in C_{pa,1}$ and $\langle f_2, p \rangle \in C_{pa,2}$. Note that the technique works even if $\rho(f_1, p) = \rho(f_2, p)$. However, if the number of elements in $\rho(f_1, p) \cap \rho(f_2, p)$ is large, it is not likely that criteria such as "number of rules" can be optimized with the aid of this technique (establishes condition 5).

3. If the two sets of constructors that are allowed at position $p$ of the constructors to be identified are identical then disjunctness can be obtained by expanding the constructors to be identified into a parent to be identified and a child which represents the identity of the parent. A useful special case of this technique is the expansion of a constructor to be identified into a new constructor with increased arity and a fixed constant at the new operand position. This technique is illustrated in chapter 10 (establishes condition 5).

4. If the constructor sets which are allowed to occur below $f_1$ and $f_2$ at a certain position other than the distinguishing position $p$ are not identical, then identity
might be obtainable by a root side identification on the constructors which make the two sets distinct. Note that a leaf side identification on these constructors is not possible because the conditions for this identification will never hold in this situation. If the constructor sets in question do not have the same number of elements then it might be possible to reduce the number of the largest set by identifying certain elements or to increase the number of the smallest set by distinguishing certain elements (establishes condition 6).

5. If the sets of constructor position pairs which can occur above \( f_1 \) and \( f_2 \) are not identical, then identity might be obtainable by a leaf side identification on the constructor-positions which make the two sets distinct. Note that a root side identification on these constructors is not possible because the conditions for this identification will never hold in this situation. If the constructor-position sets in question do not have the same number of elements then it might be possible to reduce the number of the largest set by identifying certain elements or to increase the number of the smallest set by distinguishing certain elements (establishes condition 7).

6. If the pre-condition with regard to membership of the start set (the sixth pre-condition) does not hold, then root position identification might present a solution. Suppose \( f_1 \in S \) and \( f_2 \notin S \) and \( f_1 \) and \( f_2 \) have to be identified. Two different measures are possible: a root position identification of \( f_2 \) and one specific other constructor (one for which this identification is possible, if any), or a root position distinguishing of \( f_1 \). Both measures can easily be seen to establish the sixth pre-condition of leaf side identification (establishes condition 8).

Example 9.12. An example that illustrates a measure to enable a leaf side identification. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

\[
C \supseteq \{f_1, f_2, g_1, g_2, g_3\},
\]

\[
S = \{f_1, f_2\},
\]

\[
\alpha(\{f_1, f_2, g_1, g_2, g_3\}) = \{1\},
\]

\[
\rho(f_1, 1) = \{g_1, g_2\}, \rho(f_2, 1) = \{g_2, g_3\},
\]

\[
\rho^{-1}(f_1) = \rho^{-1}(f_2),
\]

\( V, \mu \): not relevant in this example.

A leaf side identification of \( f_1 \) and \( f_2 \) becomes possible after subjecting \( g_2 \) to root side distinguishing with partition \( \{(f_1, 1)\}, \{(f_2, 1)\} \).
Root Side Identification

Five simple measures are possible to enable the root side identification transformation. The measures establish the conditions 4 through 7 of the root side identification transformation (section 6.8).

1. If the arities of the constructors to be identified are not equal then equality can be attained by increasing the arity of the lowest arity constructor or by decreasing the arity of the highest arity constructor. Both measures can be done with expansion or contraction transformations. Arity modifying transformations are discussed in detail in section 9.6.8 (establishes condition 4).

2. If the constructor sets which are allowed to occur below \( f_1 \) and \( f_2 \) at a certain position are not identical then identity might be obtainable by a root side identification on the constructors which make the two sets distinct. Note that a leaf side identification on these constructors is not possible because the conditions for this identification will never hold in this situation. If the constructor sets in question do not have the same number of elements then it might be possible to reduce the number of the largest set by identifying certain elements or to increase the number of the smallest set by distinguishing certain elements (establishes condition 5).

3. If the constructor-position pairs which can occur above \( f_1 \) and \( f_2 \) are not disjoint sets then disjointness can be obtained by leaf side distinguishing the constructors in the overlap between the two sets. If \( (c, p) \in (\rho^{-1}(f_1) \cap \rho^{-1}(f_2)) \) then \( c \) should be leaf side distinguished with distinguishing position \( p \) and the partition \( \{C_{ch,1}, C_{ch,2}\} \) of \( \rho(c, p) \) should be such that \( f_1 \in C_{ch,1} \) and \( f_2 \in C_{ch,2} \). Note that the technique works even if \( \rho^{-1}(f_1) = \rho^{-1}(f_2) \). However, if the number of elements in \( \rho^{-1}(f_1) \cap \rho^{-1}(f_2) \) is large, it is not likely that criteria such as “number of rules” can be optimized with the aid of this technique (establishes condition 6).

4. If the two sets of constructor position pairs that can occur above the constructors to be identified are identical then disjunctness can be obtained by expanding the constructors to be identified into a unary constructor (parent) and an \( n \)-ary constructor (child, \( n \) being the arity of the original constructor). This technique, which “isolates” the identity of constructors to be identified into unary constructors may sometimes enable interesting generalizations. Example 9.14 illustrates the use of this technique (establishes condition 6).

5. If the condition \( f_1 \not\in S, f_2 \not\in S \) does not hold, things can be improved by means of root position distinguishing \( f_1 \) and \( f_2 \) (establishes condition 7).

Example 9.13. An example that illustrates a measure to enable a root side identification. Consider the vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) with:

\[ C \supset \{f_1, f_2, g_1, g_2, g_3\}, \]
9.3. **RULE REDUCTIONS**

\[ S \cap \{f_1, f_2\} = \emptyset, \]
\[ \alpha(\{f_1, f_2, g_1, g_2, g_3\}) = \{1\}, \]
\[ \rho(f_1, 1) = \rho(f_2, 1), \]
\[ \rho^{-1}(f_1) \cap \rho^{-1}(f_2) = \{\langle g_2, 1 \rangle\}, \]
\[ V, \mu: \text{not relevant in this example.} \]

A root side identification of \( f_1 \) and \( f_2 \) becomes possible after leaf side distinguishing \( g_2 \) at the first (and only) operand position, with partition such that \( f_1 \in C_{ch,1} \) and \( f_2 \in C_{ch,2} \).

\[ \square \]

**Example 9.14.** An illustration of a special technique for identifications which enable generalizations. Consider the vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) with:

\[ C \supseteq \{f, f', g, g'\}, \]
\[ V \supseteq \{v, w\}, \]
\[ \rho^{-1}(f) = \rho^{-1}(f') = \rho^{-1}(g) = \rho^{-1}(g'), \]
\[ \rho(f, i) = \rho(f', i) = \rho(g, i) = \rho(g', i) \text{ for } i \in \{1, 2\}. \]

The rest of \( A \) is not relevant for this example. Consider the two rules:

\[ f(v, w) \rightarrow f'(w, v), \]
\[ g(v, w) \rightarrow g'(w, v). \]

A generalization would be possible if we could somehow identify \( f, g \) and \( f', g' \).

This is not possible in this situation because there is no way of reconstructing the original identity of the \( f, f', g, g' \) constructors. All surrounding contexts of these constructors are identical. Now consider the transformation of expanding each of \( f, f', g, g' \) into an unary constructor and below it a new binary constructor. For convenience of presentation we will denote these new binary constructors with the original names \( f, f', g, g' \). The rules become:

\[ p(f(v, w)) \rightarrow p'(f'(w, v)), \]
\[ q(g(v, w)) \rightarrow q'(g'(w, v)). \]

It is easy to see that \( p \) and \( p' \) can be identified, and also \( q \) and \( q' \), resulting in (reuse of the old names \( p \) and \( q \)):

\[ p(f(v, w)) \rightarrow p(f'(w, v)), \]
\[ q(g(v, w)) \rightarrow q(g'(w, v)). \]
The identity of the constructors $f, g$ and $f', g'$ is now if it were "isolated" in the unary constructors $p$ and $q$. An identification of $f, g$ and $f', g'$ has become possible. With $fg$ and $f'g'$ the new names the resulting rule set is:

\[ p(fg(v, w)) \to p(f'g'(w, v)), \]
\[ q(fg(v, w)) \to q(f'g'(w, v)). \]

Now a root side generalization has become possible with as final result:

\[ fg(v, w) \to f'g'(w, v). \]

This situation seems very close to the point of departure, the rules:

\[ f(v, w) \to f'(w, v), \]
\[ g(v, w) \to g'(w, v). \]

One should note however that the transformation trajecoty has had its impact on the vocabulary and on the type system and although the $p$ and $q$ constructors are not visible in the generalized rule, they are visible in terms to be rewritten (the entire transformations has definitely increased the size of some terms).

Root Position Identification.

Two simple measures are possible to enable the root position identification transformation. The measures establish the conditions 2 and 3 of the root position identification transformation (section 6.9).

- If the arities of the constructors to be identified are not equal then equality can be attained by increasing the arity of the lowest arity constructor or by decreasing the arity of the highest arity constructor. Both measures can be done with expansion or contraction transformations. Arity modifying transformations are discussed in detail in section 9.6.8 (establishes condition 2).

- If the constructor sets which are allowed to occur below a related $f_1$ and $f_2$ at a certain position are not identical then identity might be obtainable by a root side identification on the constructors which make the two sets distinct. Note that a leaf side identification on these constructors is not possible because the conditions for this identification will never hold in this situation. If the constructor sets in question do not have the same number of elements then it might be possible to reduce the number of the largest set by identifying certain elements or to increase the number of the smallest set by distinguishing certain elements (establishes condition 3).

The last four conditions of the root position identification transformation cannot be established by transformations, only by choosing the sets $F_1$ and $F_2$ well. The two measures to enable root position identification are similar to measures to enable root side identification. One is referred to the root side identification case for examples.
9.3. RULE REDUCTIONS

9.3.4 Enabling Distinguishings

Introduction

If the conditions for the leaf side distinguishing, the root side distinguishing, or the root position distinguishing transformation as described in section 6.4, 6.5, and 6.6 do not hold, it is sometimes possible to establish these conditions with one or more of the other elementary transformations form chapter 6. In the rest of this subsection will be discussed what elementary transformations can be utilized in which situation to make the conditions for the distinguishings to hold. Measures for the three distinct distinguishings will be discussed separately. The measures will be presented in an informal way as comments on the formal descriptions of the two identifications. The context for these measures in these formal descriptions can be found in section 6.4, 6.5, and 6.6. The measures are rather simple. No concrete examples will be given therefore.

Leaf Side Distinguishing

Two simple measures are possible to enable the root side distinguishing transformation.

- If the condition with regard to the membership of $S$ of the root constructors of rule patterns does not hold then a root position distinguishing transformation might present a solution.

- If the rule set contains collapsing rules, they can be eliminated by root side or leaf side specializations.

Root Side Distinguishing

Two simple measures are possible to enable the root side distinguishing transformation.

- If the constructor to be distinguished is a member of $S$ then a root position distinguishing transformation might present a solution.

- If the rule set contains collapsing rules, they can be eliminated by root side or leaf side specializations.

Root Position Distinguishing

One simple measure is possible to enable the root side distinguishing transformation.

- If the rule set contains collapsing rules, they can be eliminated by root side or leaf side specialization.
9.3.5 Reducing Rule Increases

Introduction

In section 9.3.2 it was shown that generalizations can be made possible by well-chosen distinguishing or identification transformations. Unfortunately these transformations sometimes have the consequence of increasing the number of rules; conditions under which this will happen were stated in section 9.3.1. The increase associated with the distinguishing or identification may outweigh the decrease associated with the generalization. There are simple techniques to reduce the amount of increase associated with root side distinguishing and leaf side distinguishing transformations. The technique is in essence the application of another generalization, immediately after the distinguishing or identification. With the root position distinguishing transformation the worst case increase in rule numbers is less pronounced than with the other distinguishing transformations. There is a maximum of one additional rule per constructor to be distinguished. No techniques are known to reduce this number.

Leaf Side Distinguishing

The increase of rule numbers due to this transformation is not entirely visible in the algorithm for the derivation of the rule set (section 6.4) because the algorithm utilizes the possibility of non-maximally typed variables. Cascaded transformations require often maximally typed variables (at intermediate stages) so figures for rule increases are based on transformations that are enriched with measures to obtain maximally typed variables.

The factor $|\rho(f, p)|^n$ in the list of section 9.3.1 can in most cases be reduced to a factor $2^n$, gaining $|\rho(f, p)|^n - 2^n$ rules. With leaf side distinguishing the factor $|\rho(f, p)|$ appears in the case of rules with occurrences of $f$ with a variable at position $p$ only in the right hand side pattern of the rule. The number of additional rules can be reduced if at least one constructor with $v$ at $p$ in the left hand side of the pattern is also subjected to leaf side distinguishing, with the same partition of $\rho(f, p)$. The gain is reduced if this particular constructor occurs with a variable at position $p$ in other rules, but the net effect may be positive, even after cascaded applications of this technique over several rules. The gain is due to an easy to perform generalization that becomes possible after the "increase reducing" distinguishing.

Example 9.15. An example that illustrates a measure to reduce the amount of extra rules with leaf side distinguishing. Consider the vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

\[
C \ni \{f, g, h, a_1, a_2, a_3, a_4\},
\]

\[
V \ni \{v, \ldots\},
\]

\[
S = \{f\},
\]

\[
\alpha(\{f, g, h\}) = \{1\}, \alpha(\{a_1, \ldots, a_4\}) = \{0\},
\]
\[ \rho(f,1) = \{ C \}, \]
\[ \rho(\{g,h\},1) = \{ a_1, \ldots, a_4 \}, \]
and consider the rewrite system consisting of the single rule:
\[ f(g(v)) \to f(h(v)). \]
Subjecting the entire system to leaf side distinguishing \( h \) with partition \( \{ \{a_1, a_2\}, \{a_3, a_4\} \} \)
of \( \rho(h,1) \) leads to a rule set with four rules:
\[ \{ f(g(a_1)) \to f(h_1(a_1)), \]
\[ f(g(a_2)) \to f(h_1(a_2)), \]
\[ f(g(a_3)) \to f(h_2(a_3)), \]
\[ f(g(a_4)) \to f(h_2(a_4)) \}. \]
Performing a leaf side distinguishing of \( g \) on these four rules with the same partition gives possibilities for leaf side generalizations. The resulting rule set can be:
\[ \{ f(g_1(v)) \to f(h_1(v)), \]
\[ f(g_2(v)) \to f(h_2(v))) \}, \]
which is substantially better than the four rules with a total pattern size of 24 constructors. If the distinguishing of \( g \) is performed first the rule set never reaches the extent of four rules.

\[ \square \]

**Root Side Distinguishing**

The factor \( |\rho^{-1}(f)| \) in the list of section 9.3.1 can in most cases be reduced to a factor 2, gaining \( |\rho^{-1}(f)| - 2 \) rules. With root side distinguishing the factor \( |\rho^{-1}(f)| \) appears in the case of rules with a right hand side root equal to \( f \): the constructor to be distinguished and with a left hand side root not equal to \( f \). The number of additional rules can be reduced if the left hand side root is also subjected to root side distinguishing, with the same bi-partition of parent-position pairs. The gain is reduced if this left hand side root occurs at the root position of other rules, but the net effect may be positive, even after cascaded applications of this technique over several rules. The gain is due to an easy to perform generalization that becomes possible after two or more root side distinctions.

**Example 9.16.** An example that illustrates a measure to reduce the amount of extra rules with root side distinguishing. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:
\[ C \supset \{ f_1, f_2, f_3, f_4, g, h, a \}, \]
\[ V \supset \{ v, \ldots \}, \]
\[ S = \{ f_1, \ldots, f_4, a \}, \]
\[ \alpha(\{f_1, \ldots, f_4, g, h\}) = \{1\}, \alpha(a) = 0, \]
\[ \rho(\{f_1, \ldots, f_4\}, 1) = \{C\}, \]
\[ \rho(\{g, h\}, 1) = \{f_1, \ldots, f_4, a\}, \]

and consider the rewrite system consisting of the single rule:

\[ g(a) \to h(a). \]

Subjecting the entire system to root side distinguishing \( h \) with partition \( \{\{f_1, f_2\}, \{f_3, f_4\}\} \) of \( \rho^{-1}(h) \) leads to a rule set with four rules:

\[ \{ f_1(g(a)) \to f_1(h_1(a)), \]
\[ f_2(g(a)) \to f_2(h_1(a)), \]
\[ f_3(g(a)) \to f_3(h_2(a)), \]
\[ f_4(g(a)) \to f_4(h_2(a)) \}. \]

Performing a root side distinguishing of \( g \) on these four rules with the same partition gives possibilities for root side generalizations. The resulting rule set can be:

\[ \{ g_1(a) \to h_1(a), \]
\[ g_2(a) \to h_2(a) \}, \]

which is substantially better than the four rules with a total patterns size of 24 constructors. If the distinguishing of \( g \) is performed first the rule set never reaches the extent of four rules.

\[ \square \]

### 9.3.6 A Simple Strategy

It is easy to design a simple strategy for the reduction of rule numbers with the presented simple techniques. A summary of the techniques presented so far:

- enabling generalizations,
- enabling identifications,
- enabling distinguishings,
- reducing rule increases.
Generalizations are the only means to achieve rule reductions and rule subsets are possible candidates for generalizations. An algorithm for the reduction of rule numbers should therefore consider all subsets of a certain rule-set and determine whether or not a generalization is possible or can be made possible with one or more elementary transformations. The power-set of a rule-set tends to be large, so intelligent techniques need to be deployed to control this exponential number of subsets. These techniques can for instance be based on the observation that if a subset $S_x$ of the rule-set is not generalizable, then supersets of $S_x$ are mostly not generalizable as well. In the rest of this subsection we consider the strategy below the level of the iteration over the rule subsets.

Each rule subset corresponds to a tree of measures (transformations) to enable a generalization. A tree because at most stages in the process to make a generalization possible there are multiple alternatives. The root of the tree represents the state in which a generalization on a certain rule subset is not yet possible, the edges in the tree represent elementary transformations as described in previous sections, and the leaves in the tree represent states in which a generalization is possible or states in which a generalization is not possible and also no measures to enable a generalization. We will call these distinct kinds of leaves white leaves (generalization possible) and black leaves (generalization not possible). A path from root to a white leave corresponds to a sequence of elementary transformations which enables a generalization. The problem whether or not there is a finite sequence of transformations that enables a generalization is likely to be undecidable (additional research is necessary for that, see section 11.4). In that case there is in general no number $n$ (dependent on the particular rule subset) such that the non-existence of a white leaf is implied by the non-existence of such a leaf at the end of a path with length less than $n$. Note that sheer decidability is not enough in practice. What we want is a sufficiently small number $n$. If we assume un-decidability then precautions have to be taken in an algorithm to explore this search space.

The objective is to reduce the number of rules in a rewrite system by a generalization. The path to this goal consists of zero, one or more elementary transformations. The generalization gives a reduction in the number of rules, all other transformations will preserve the number of rules, increase it by a certain amount, or decrease it (only the distinguishing transformations of section 9.3.5). Summarizing: a generalization corresponds to a known decrease of one or more rules and an elementary transformation corresponds to a known increase or decrease of numbers. With this knowledge it is possible to attribute the search tree of transformations and to associate a natural number that reflects the amount of reduction to each of the nodes of the tree. If the generalization replaces $n$ rules by one rule then the reduction associated with the generalization is $n-1$ and the reduction attribute of the root node should be $n-1$. The attributes of descendant nodes can be calculated by subtracting the increase (a decrease is simply a negative increase) from the attribute of the ancestor node.

We can say now that we have to search for a white leaf with a positive valued attribute. The search space will in practice be small. The reason is the following.
The global tendency of attribute values is decreasing. Increasing steps are possible (by the increase reducing distinguishing of section 9.3.5) but the increase will never be larger than the decrease of the previous step. This means that occurrences of positive valued attributes become less likely at deeper positions in the search tree (attributes of nodes farther away from the root of the search tree). This in its turn, means that it is possible to minimize the probability of overlooking a possibility to reduce the number of rules by taking a large enough depth to descend in the search tree.

The principles of an algorithm to find rule number reducing transformation sequences are fixed by the above informal description. A more formal representation of this algorithm (for instance in pseudo code) seems not appropriate here, as there are still unanswered questions and details to be worked out in further research.

- Is the problem whether or not there is a rule reducing transformation sequence undecidable?
- What is an efficient strategy to consider rule subsets for generalization?

A remark with regard to the last question: if a rule subset $S$ does not correspond to a search tree with white leaves, then supersets of $S$ need not to be taken in consideration, however if a rule subset $S$ corresponds to a search tree with white leaves with negative or zero valued attributes, then supersets of $S$ might lead to rule reductions because the attribute of the root node of the search tree is larger for larger rule subsets.

### 9.3.7 Practical Benefits of Rule Reductions

A few remarks will be made on the practical benefits of rule reductions. Intuitively: rewrite systems with fewer and simpler rules perform better. It is an interesting question if it is possible to somehow quantify this improved performance. We will informally discuss the performance issue with regard to two well-known matching algorithms: simple tree matching and bottom-up tree matching.

The performance of an implementation of a subtree replacement system (a term rewriting system for instance) depends heavily on the performance of its matching algorithms since each replacement action is preceded by a matching action: a search for a pattern to be replaced. We assume a template replacement system to globally consist of two parts: the matching algorithm (the algorithm that searches for occurrences of patterns) and the replacement algorithm (the algorithm that replaces found patterns by other patterns). We will consider two different matching algorithms: simple tree matching, which is a simple search for pattern occurrences in the tree (term) to be rewritten, one pattern at a time [32], and bottom-up tree matching, which is a kind of compiled tree matching with an automaton which locates all pattern occurrences in the tree (term) to be rewritten in one pass [32, 1, 11].

It is not difficult to see that when using the simple tree matching algorithm, a reduction of the number of rules will in general result in a better performance. Fewer
rules means fewer patterns to match, which is equivalent to a reduced workload for the matching algorithm. Besides that, rule reductions which are attained with mere generalizations also correspond to decreased pattern sizes (section 9.2), which is also beneficial with regard to performance, both of matching algorithms and of replacement algorithms. Without formal definition of time and space performance of algorithms we state that, with a simple tree matching algorithm, a reduction of the number of rules by a factor \( r \), the time performance of the entire subtree replacement system will be improved by at least a factor \( r \) (for reasonable measures for this performance).

When using the compiled bottom-up tree matching algorithm the time performance benefits are less pronounced. The time performance of the bottom-up matching algorithm does not depend on the number of rules \([32]\), so the reduction of the number of rules does not seem something to strive for. Fewer rules correspond to less complex matching automatons with fewer states \([32]\), but the time performance of these automatons is not affected directly. However, the replacement algorithm does benefit from the decreased pattern sizes which usually accompany rule reductions. Moreover, it will always be advantageous to keep the number of states of the matching automaton as low as possible (the number of states of bottom-up tree matching algorithms tends to be high \([32, 10]\)). Reducing the number of states of the matching automaton will keep the size of the transition tables of the matching automaton manageable. Finally: there is a pronounced performance benefit if rule reductions can keep the number of states of a matching automaton below a number which can be encoded by commonly used word sizes of computers: one byte \((2^8\) states\) or two bytes \((2^{16}\) states\).

It is difficult to say much more about the relation between rule reductions and performance of concrete subtree replacement systems. More practically oriented research has to be done for that. Therefore the transformation algorithms first have to be implemented. With implemented transformation algorithms and an implementation of the strategies which are discussed in this chapter it is possible to transform large transformation systems and to determine the effect of these transformations on performance by experiment. In chapter 11 some remarks are made on the implementation of the developed theory.

Besides being beneficial for the performance of implementations of subtree replacement systems, rule reductions can also serve a different purpose. Rule reductions, especially rule reductions which can be obtained by generalizations which need few or no additional transformations at all to become possible, will remove redundant information from the rule specifications. If it is possible to reduce the number of rules then the original rule specification has a complexity which is not inherent in the rewrite problem that is solved by the rule specification. In that case it is desirable to reduce the complexity for a number of reasons besides an eventual performance benefit. A less complex system is for instance easier to understand, to communicate and to maintain. It should be noted that not all complexity reductions enhance readability. Examples of situations in which this is not the case can be found in chapter 10.
9.4 Size Reductions

9.4.1 Introduction

With size reductions we mean a transformation, or a sequence of transformations, which reduces the number of constructor occurrences in some or all terms of a term set. We will only consider transformations which lead to direct and unambiguous reductions. Transformations with which the size of some terms is increased and of other terms decreased, and which are beneficial because for certain reasons the decrease outweighs the increase, are not considered.

There is only one elementary transformation with which size reductions can be attained: contraction. The contraction transformation will in general increase both the number of rules and the number of constructors of the vocabulary, so size reductions are in general conflicting with rule reductions and with vocabulary reductions. However, there are situations in which increases in the number of rules or the number of constructors can be avoided, or restricted.

9.4.2 Size Reductions Versus Vocabulary Reductions

The contraction transformation will in general increase the number of constructors by one. The number of constructors remains the same if either the pa or the ch constructor (section 6.10) can be eliminated because it has become non-integrated or non-satisfiable (section 5.2). The number of constructors is reduced by one if both the pa and the ch constructors can be eliminated because of non-integratedness or non-satisfiability. It is not difficult to formulate a simple strategy for size reductions now.

- If the vocabulary is not allowed to grow, the contractions have to be restricted to the situations mentioned above.

- If the growth of the vocabulary has to be kept to a minimum, the contractions of the situations mentioned above should be part of the sequence of contractions anyway.

- If there is no restriction with regard to the size of the vocabulary, any term can be reduced to a single constant. Note that the size of terms is in general not bounded, so there is not a finite sequence of contractions that will reduce each term of a vocabulary with unbounded term sizes to a single constant. However, for each term of such a vocabulary there is a finite sequence of contractions that will reduce it to a single constant. The restriction on the contraction of two identical constructors is no problem because any ground term consisting of two or more different constructors, so there is always a pair which can be contracted. For instance, the term \( f(f(f(f(a)))) \) can be contracted as:

\[
f(f(f(f(a)))) \Rightarrow \]

9.4. SIZE REDUCTIONS

\[ f(f(f(a')) \Rightarrow \]
\[ f(f(a'')) \Rightarrow \]
\[ f(a''') \Rightarrow \]
\[ a''' \].

An alternative direction for collapsing this term is possible by first performing a root position distinguishing transformation (section 6.6) on the root occurrence of \( f \).

A lot more can be said about the efficiency of size reductions by contractions which correspond to growing vocabularies. Some constructors will occur more often in terms than other constructors. Contracting constructor pairs which occur more often is more efficient with regard to the penalty of the increased vocabulary size.

Calculations aimed at the determination which constructor pairs occur most often in practice should be based on a probability distribution for term occurrences. Not all well-formed terms of a certain term set have an equal occurrence probability in practice. If the term set is infinite, which is usually the case, then in practice either:

- terms with size larger than a certain bound have a zero occurrence probability or:
- terms have an occurrence probability which decreases exponentially with their size.

In fact there is not much difference between the two because an exponential decreasing probability means a negligible probability for large terms.

If a probability distribution is known, say \( P : T \rightarrow [0, 1] \), then relative occurrence number of constructor pairs \( pa - ch \) (in which the position \( p \) at which \( ch \) occurs is relevant) can be calculated by adding the products of occurrence probability of all terms with the occurrence number of the particular pair in the particular term, i.e.

\[ \sum_{t \in T} P(t) \times \#((pa, ch, p), t), \]

with \( \#((pa, ch, p), t) \) the number of combined occurrences of \( pa \) and \( ch \) at position \( p \) in \( t \) (called expectation in the field of statistics). Contracting the combinations with the highest relative occurrence numbers is most efficient with regard to the penalty of increased vocabulary size.

9.4.3 Size Reductions Versus Rule Reductions

With regard to the growth of the rule set the following can be said. Contraction transformations will in general result in rule multiplications (section 6.10). There are however situations in which the growth is small or even zero. Concrete figures are presented in section 9.3.1.
The relation between size reductions and rule number increases is more complicated than the relation between size reductions and vocabulary increases because of the involvement of the rule set. Like with the comparison between size reductions and vocabulary reductions there are contractions that are more efficient with regard to the penalty of increased rule numbers. This efficiency is, amongst others, related to the occurrence of the constructors to be contracted in patterns of rules.

### 9.4.4 Practical Benefits of Size Reductions

A few remarks will be made on the practical benefits of size reductions. Intuitively: rewrite systems with smaller terms have at least a better space performance. Smaller terms will occupy less space in memory. Smaller terms will usually also correspond to a better time performance, however this will not always be the case if the transformation that accomplishes the size reductions do increase the number of rules.

### 9.5 Vocabulary Reductions

#### 9.5.1 Introduction

With vocabulary reductions we mean a transformation, or a sequence of transformations, which reduces the total number of constructors in the constructor set \( C \) of the vocabulary. In this section a simple strategy to attain vocabulary reductions will be presented.

#### 9.5.2 A Simple Strategy

Identifications and contractions are the transformations with which vocabulary reductions can be attained. Unfortunately each of the three identification transformations has quite strong pre-conditions and the conditions under which a contraction results in the reduction of the volume of the vocabulary rarely occur in practice. If identifications are possible, they should be carried through anyway. Identifications will not affect the size of terms and occasionally increase the number of rules (section 9.3.1). If it is possible to reduce the size of the vocabulary by a contraction, this should be done anyway. Vocabulary reduction by contraction is possible if and only if the \( pa \) constructor can only occur with \( ch \) at position \( p \) and \( ch \) only below \( pa \) at \( p \) (\( pa, ch, \) and \( p \) are defined in section 6.10). Vocabulary reduction by contraction will occasionally increase the number of rules.

If the pre-conditions for identification or contraction do not hold there are more complex strategies to achieve reductions in the size of the vocabulary, strategies which are based on the restructuring domain mapping concept (section 8.3). It is often possible to replace a set of combinations of constructors by a different set of combinations of constructors, with which the total number of involved constructors is reduced. The most simple instance of this technique is the replacement of a set of
single constructors with a combination of two constructors. Consider the following example for an illustration of this technique.

**Example 9.17.** Shrinking of vocabulary size by replacing single constructors by combination of constructors. Let $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ a vocabulary with:

$C = \{f_1, \ldots, f_{16}, a\}$,

$S = \{f_1, \ldots, f_{16}\}$,

$\alpha(\{f_1, \ldots, f_{16}\}) = \{1\}, \alpha(a) = 0$,

$\rho(f_i, 1) = \{f_1, \ldots, f_{16}, a\}$ for all $1 \leq i \leq 16$,

$V, \mu$: not relevant.

The size of $C$ can be reduced from 17 to 9 by replacing each of $f_1, \ldots, f_{16}$ by a combination of a constructor from $\{g_1, \ldots, g_4\}$ and one from $\{h_1, \ldots, h_4\}$. Since $4 \times 4 = 16$ and $4 + 4 = 8$ there is a reduction of $16 - 8 = 8$ constructors. The new vocabulary $A' = \langle C', V', S', \alpha', \rho', \mu' \rangle$ becomes:

$C' = \{g_1, \ldots, g_4, h_1, \ldots, h_4, a\}$,

$S' = \{g_1, \ldots, g_4\}$,

$\alpha'(\{g_1, \ldots, h_4\}) = \{1\}, \alpha'(a) = 0$,

$\rho'(g_i, 1) = \{h_1, \ldots, h_4\}$, for all $1 \leq i \leq 4$,

$\rho'(h_i, 1) = \{g_1, \ldots, g_4, a\}$, for all $1 \leq i \leq 4$,

$V', \mu'$: not relevant.

It is easy to see that there is a finitely based domain mapping between $\mathcal{TG}_A$ and $\mathcal{TG}_{A'}$ with $\psi$ as follows:

$\psi(a) = a$,

$\psi(f_1(\Box)) = g_1(h_1(\Box))$,

$\psi(f_2(\Box)) = g_1(h_2(\Box))$,

$\vdots$

$\psi(f_{16}(\Box)) = g_4(h_4(\Box))$.  \qed
Obviously this strategy will increase the size of terms. Example 9.17 illustrates a special case of the more general technique of achieving reductions by replacing combinations of constructors by other combinations. An example of the more general technique is the replacement of combinations of constructors from \( \{g_1, g_2\} \) and \( \{h_1, \ldots, h_8\} \) by constructors from \( \{g_1, \ldots, g_4\} \) and \( \{h_1, \ldots, h_4\} \), with which a reduction of \((2 + 8) - (4 + 4) = 2\) can be achieved. The presented examples are all based on manipulations with unary constructors, to keep matters simple. Constructors with arity larger than one will in general provide more opportunities to shrink vocabularies.

### 9.5.3 Practical Benefits of Vocabulary Reductions

There is one case in which a vocabulary reduction can have a pronounced effect on the space performance (and sometimes also on the time performance). That is the case when a vocabulary reduction can reduce the number of constructors just below a number which can be encoded with commonly used word sizes in computers: one byte \( (2^8 \text{ constructors}) \) or two bytes \( (2^{16} \text{ constructors}) \). If it is possible to save one byte on each encoded constructor the total amount of reduction will be substantial.

### 9.6 Acquiring Special Properties

#### 9.6.1 Introduction

In this section a number of transformations will be presented that establish properties which are likely to be useful. The first five subsections are devoted to transformations which establish the properties that were defined in definition 5.9 through 5.13. The remaining subsections deal with properties with regard to modular decomposability of rule sets and properties with regard to the arity of constructors.

#### 9.6.2 Acquiring Integratedness

According to definition 5.9 a vocabulary is integrated if and only if all constructors are integrated, the set of integrated constructors being defined recursively as the set of constructors from \( S \) and the constructors that appear in some \( \rho \)-value of a integrated constructor. The elimination of non-integrated constructors from a vocabulary can be done straightforwardly. They simply have to be removed from the vocabulary in a consistent manner. The domain mapping can be the function \( id \) (the transformation is another useful instance of an identical domain transformation).

**Scheme 9.18.** A general scheme for a transformation which eliminates the set of non-integrated constructors in a consistent way from vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \). The vocabulary \( A'' = (C'', V'', S'', \alpha'', \rho'', \mu'') \), which will be defined in terms of \( A \), can easily be shown to generate the same set of well-formed terms, i.e. \( T_{A''} = T_A \).

Let \( C_{\text{non}} \subseteq C \) be the subset of all constructors that are not integrated. The set of non-integrated constructors \( C_{\text{non}} \) should be removed from \( A \) as follows:
9.6. ACQUIRING SPECIAL PROPERTIES

- \( C'' = C - C_{\text{non}} \);
- \( V'' = V - \{ v \in V \mid \mu(v) \cap C_{\text{non}} \neq \emptyset \} \);
- \( S'' = S - C_{\text{non}} \);
- \( \alpha'' = \alpha \uparrow \{ c \leftarrow \perp \mid c \in C_{\text{non}} \} \);
- \( \rho'' = \rho \uparrow \{ \langle c, i \rangle \leftarrow \perp \mid c \in C_{\text{non}}, 1 \leq i \leq \alpha(c) \} \);
- \( \mu'' = \mu \uparrow \{ v \leftarrow \perp \mid v \in V, \mu(v) \cap C_{\text{non}} \neq \emptyset \} \).

It is easy to see that the removal of non-integrated constructors will not introduce non-satisfiability (section 9.6.3), trivial combinations (section 9.6.4), nor will it destroy any of the other properties: being normalized and uniquely starting. The fact that no non-satisfiable constructors will be introduced is important since the removal of non-satisfiable constructors will occasionally introduce non-integrated constructors, so termination is at stake. To see that no non-satisfiability will be introduced, note that none of the constructors that will be removed by scheme 9.16 occurs in any of the \( \rho \)-values of the constructors that remain (otherwise these constructors would not have been non-integrated).

9.6.3 Acquiring Satisfiability

According to definition 5.10 a vocabulary is satisfiable if and only if each individual constructor is satisfiable, the set of satisfiable constructors being defined recursively as the set of constants and the constructors that can receive at least one satisfiable constructor at each operand position. The elimination of non-satisfiable constructors from a vocabulary can be done straightforwardly. They simply have to be removed from the vocabulary in a consistent manner. The domain mapping can be the function \( \text{id} \) (the transformation is another useful instance of an identical domain transformation). Non-satisfiable constructors can also be removed with scheme 9.16. If \( C_{\text{non}} \) in this scheme is defined to be the set of non-satisfiable constructors then scheme 9.16 works well to acquire satisfiability.

The removal of non-satisfiable constructors will occasionally introduce non-integrated constructors. Consider for instance a binary constructor \( f \) with \( \rho(f, 1) = \emptyset, \rho(f, 2) \ni g \), and \( \rho^{-1}(g) = \{ \langle f, 2 \rangle \} \). If \( f \in S \) then \( f \) and \( g \) are reachable. Removal of the non-satisfiable \( f \) will leave \( g \) non-integrated (provided that \( g \notin S \)). If there are both non-integrated and non-satisfiable constructors in the vocabulary, the non-satisfiable constructors should be removed first therefore. The process of removing non-satisfiable and non-integrated constructors will always terminate (section 9.6.2).

9.6.4 Acquiring the Trivial Combination Free Property

According to definition 5.11 a vocabulary is trivial combination free if and only if the cardinality of the function values of \( \rho \) is always greater than one. The
elimination of empty set values of \( \rho \) was subject of section 9.6.3, so it is possible to reach a state in which a \( \rho \)-value has one element, if it has less than two elements. The elimination of singleton set \( \rho \)-values can be done with the contraction transformation. If constructor \( c \) has a singleton set \( \rho \)-value at position \( p \) and \( \rho(pa, p) = \{ ch \} \) then \( pa \) and \( ch \) should be contracted in the way as defined in section 6.10. The elimination of trivial combinations is not an identical domain transformation; the domain mapping can be found in section 6.10.

Note that it may require more than one contraction to remove a singleton set from the set \( YC \), in terms of which trivial-combination-freeness was defined, because the same singleton set may occur more than once as \( \rho \)-value.

Note also that the contraction transformation may result in non-integratedness and/or non-satisfiability. Therefore, if a vocabulary contains both trivial combinations, non-satisfiable and/or non-reachable constructors, the trivial combinations should be removed first.

### 9.6.5 Normalization

According to definition 5.12 a vocabulary is normalized if and only if the set of represented types in the variables \( YV \) do not exhibit proper overlap among its elements. If this property doesn't hold, normalization can proceed as follows. Let \( P \) be the coarsest partition of \( C \) such that:

\[
\forall p \in P, yv \in YV \ [p \cap yv = \emptyset \lor p \subseteq yv].
\]

It is easy to see that such a \( P \) exists for any vocabulary, and is unique for any vocabulary. This \( P \) induces a partition on each \( \rho \)-value. Now if each constructor of \( C \) is subjected to generalized leaf side distinguishing (section 8.4) for each position and with the partition that is induced by \( P \), then the resulting vocabulary is normalized. A proof of this is almost trivial: distinguishing transformations will never introduce proper overlap or properly including overlap, while existing proper overlap and properly including overlap is eliminated by the described distinguishing transformations.

### 9.6.6 Acquiring the Uniquely Starting Property

According to definition 5.13 a vocabulary is uniquely starting if and only if constructors which are allowed as root constructor are not allowed inside terms and vice versa. In the terminology of section 6.3.2: each constructor has to be specific. Constructors which are not specific can be made so by the root position distinguishing transformation (section 6.6). All constructors which are non-specific have to be distinguished in connection with root position (simultaneously). It is easy to verify that the conditions of this transformation are satisfied if all non-specific constructors are distinguished simultaneously. Also in this case the domain mapping is not equal to \( id \); the domain mapping is also presented in section 6.6.
9.6. ACQUIRING SPECIAL PROPERTIES

9.6.7 Acquiring Modular Decomposability

The subject of modular properties of term rewriting systems has received quite a lot of attention in recent years [46, 58, 45, 57]. Properties of rewrite systems like confluence and termination, which are sometimes difficult to prove, are modular, or modular under certain additional conditions. Modularity means that if the rule set can be partitioned into two or more subsets, and if the patterns of these rules do not share constructors over the boundaries of the rule subsets, then the entire system has a certain modular property if and only if each of the subsystems has the modular property. Confluence is a modular property [58] and termination is a modular property if non of the rules of the entire system is collapsing or if non of the rules of the entire system is duplicating [53]. Modular decomposability eases provability of important properties like confluence and termination.

The disjunctness requirement of constructor subsets is quite a strong requirement. The question if this requirement could somehow be relaxed was addressed in [61]. The elementary transformations of chapter 6 will be shown to be a very simple means to make this requirement to hold in many interesting cases. A few simple examples to illustrate the possibility to fulfill the disjunctness requirement for modular properties.

Example 9.19. Lifting to modularity. Consider vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

$C = \{f, g, h, a\}$,
$V = \{v, \ldots\}$,
$S = \{f, g, a\}$,
$\alpha(\{f, g, h\}) = \{1\}, \alpha(a) = 0$,
$\rho(f, 1) = \rho(g, 1) = \{f, g, h, a\}$,
$\rho(h, 1) = \{f, g, a\}$,
$\mu(v) = \{f, g, a\}$.

The rule set:

$\{f(h(v)) \rightarrow f(v), g(h(v)) \rightarrow g(v)\}$

is easily be seen to be confluent, terminating, and non-collapsing. However, confluence and termination cannot be inferred from the confluence and termination of both individual rules because they share the constructor $h$. Root side distinguishing $h$ with partition $\{\{(f, 1)\}, \{(g, 1)\}\}$ of $\rho^{-1}(h)$ leads to the rule-set:

$\{f(h_1(v)) \rightarrow f(v), g(h_2(v)) \rightarrow g(v)\},$
in which both individual rules do not share constructors. By the modularity of
confluence and the restricted modularity of termination, confluence and termination
can now be inferred for the entire system.

A more complex example is the following one, in which distinguishing leads to
rule multiplication. Consider the vocabulary \( A = (C, V, S, \alpha, \rho, \mu) \) with:

- \( C = \{c_1, c_2, f, g, h, a\} \),
- \( V = \{v, \ldots\} \),
- \( S = \{c_1, c_2\} \),
- \( \alpha(\{c_1, c_2, f, g, h\}) = \{1\}, \alpha(a) = 0 \),
- \( \rho(c_1, 1) = \rho(c_2, 1) = \{f, g, h\} \),
- \( \rho(f, 1) = \rho(g, 1) = \rho(h, 1) = \mu(v) = C \).

The rule set:

\[
\{c_1(f(v)) \rightarrow c_1(g(v)),
\quad g(v) \rightarrow h(v),
\quad c_2(h(v)) \rightarrow c_2(f(v))\},
\]

is easily seen to be terminating, in spite of the cycle \( f(v) \rightarrow g(v), g(v) \rightarrow h(v), h(v) \rightarrow f(v) \), which seems to be present in the rules. Although the rule set does not contain
collapsing rules, there is no partition of this rule set into two or three subsets from
which termination for the entire system can be inferred. Root side distinguishing
\( f, g, h \) with partition \( \{\{c_1, 1\}\}, \{\{c_2, 1\}\} \) leads to the rule set:

\[
\{c_1(f_1(v)) \rightarrow c_1(g_1(v)),
\quad g_1(v) \rightarrow h_1(v),
\quad g_2(v) \rightarrow h_2(v),
\quad c_2(h_2(v)) \rightarrow c_2(f_2(v))\},
\]

which allows a partition for which the disjunctness requirement holds, namely:

\[
\{\{c_1(f_1(v)) \rightarrow c_1(g_1(v)),
\quad g_1(v) \rightarrow h_1(v) \},
\quad \{g_2(v) \rightarrow h_2(v),
\quad c_2(h_2(v)) \rightarrow c_2(f_2(v))\}\}.
\]

The two subsystems are obviously terminating, so termination of the entire
system can be inferred. Both subsystems consist of two rules, which do not have more
constructors than the rules of the original system. This means that both subsystems
are (at least in some sense) simpler than the original system of three rules.
9.6.8 Acquiring Properties with Regard to Arity

It is relatively easy to change the arity of one or more constructors. Arity changing transformations can be used to obtain properties such as:

- no constructors with arity greater than 2,
- constructor $c$ needs to be replaced by a constructor $c'$ with a different arity,
- no constructors with arity 1,
- etc.

Virtually all arity changing transformation targets can be reached by the expansion transformation (section 6.11).

- A constructor can be replaced by a constructor with an arity which is one larger than the original arity by generating a fixed constant at the $n+1$-th operand position.

- The arity of a constructor can be reduced if it is larger than 2 by expanding it into two constructors that both have an arity which is 2 or larger than 2.

With the second technique the sum of the arities of the new constructors is one larger than the original arity, so it is easy to see that no reduction is achieved if one of the new constructors has arity one.

The arity changing transformations are simple and do not have a serious impact on the rules of a rewrite system because the expansion transformation will never cause rule multiplications.

Contraction transformations can also be used to reach a specific goal with regard to arity. However, it is in general more difficult to reach a certain predefined target with contraction transformations, because the arity of the result of a contraction depends entirely on the arities of the involved constructors.

Example 9.20. Arity transformations with expansion and contraction. Consider the vocabulary $A = \langle C, V, S, \alpha, \rho, \mu \rangle$ with:

$C = \{f, g, h, a\},$

$\alpha(\{f, g, h\}) = \{2\}, \alpha(a) = 0,$

$\rho(f, 2) = \{f, g, h\},$

$\rho(g, 2) = \{f, g, h\},$

rest of $A$ not relevant for this example.
Suppose that we want to have a term set in which the root constructor of each term is ternary, i.e. has arity three. Something has to be done with \( f \) and \( g \) to achieve this goal, because these are the constructors that can act as root of well-formed terms.

A solution with expansion transformations is to expand each occurrence of \( f \) and each occurrence of \( g \) into a new constructor \( (f' \text{ for } f \text{ and } g' \text{ for } g) \) with a fixed constant at a new (third) operand position. The extent of this measure can be restricted to the root constructor of terms by first performing a root position distinguishing transformation on \( f \) and \( g \) (this is not necessary however).

A solution with contraction transformations is to contract occurrences of \( f \) and \( g \) with each of its possible second operands. The allowed constructors at the second position of \( f \) and \( g \) are \{\( f, g, h \)\}, each of which has arity two. So contraction will effectively increment the arity of \( f \) and \( g \) by one. Because contraction of two identical constructors \( (f, f \text{ and } g, g) \) is not possible, a root position distinguishing transformation on \( f \) and \( g \) has to be performed first in this case. \( \square \)

There are lots of practical uses of the arity changing transformations. Sometimes it is necessary to keep the arity of constructors below a certain fixed number. Making the arity of two constructors equal was earlier mentioned as a transformation that could make identifications possible (section 9.3.3). It is difficult to find practical examples which do not look trivial; no concrete examples will be given therefore. The whole subject of properties with regard to arity was included to show that these simple transformations are also formalized by our theory.
Chapter 10

Case

10.1 Introduction

In this chapter a case will be presented to demonstrate the practical applicability of the developed theory. It was difficult to find a suitable case for the following reasons.

- The theory is developed for ordinary typed rewriting, while most existing rewrite specifications are conditional rewrite systems or conditional typed rewrite systems. There seems no insurmountable barrier to generalize the theory to conditional typed rewrite systems, but additional research is necessary (various directions of further research are discussed in chapter 11).

- The theory works best on large rewrite specifications because of the following.

  - Large specifications tend to have large sophisticated type systems (many types, many relations between types). Conditions for the optimizations as discussed in chapter 8 generally occur only in specifications with large type systems.

  - The optimizations as discussed in chapter 8 are rather trivial for small rewrite specifications, therefore these optimizations are mostly carried through manually during the design of the specification. Small rewrite specifications in which these optimizations can be done look therefore artificial or suggest bad design. However, the optimizations are far less trivial for large specifications. For large rewrite specifications one has in fact the choice between an easy to write and easy to understand specification style and a style which is minimal or optimal in certain respects (e.g. number of rules, sizes of terms, size of vocabulary, etc.).

The case we have chosen is an algebraic specification for the static semantics of Pascal in the ASF+SDF formalism [30]. This case is a compromise in certain respects and has the following problems.
CHAPTER 10. CASE

- It is written in ASF+SDF, a specification language with conditional rules (equations, as they are called in ASF+SDF) and sequence (list) primitives. The generalization of our theory to conditional rules is summarily discussed in chapter 11, the generalization to a formalism with sequence primitives is entirely left for further research. However, the additional features of ASF+SDF pose little problems. There are lots of opportunities for optimizations in the specification which do not interfere with conditional rules or sequence constructions (many of the rules have empty condition parts and do not use the sequence primitives). Sometimes it is also easy to transform uses of the more advanced primitives into constructions within the bounds of ordinary typed rewriting.

- The specification is large: 65 pages in the original technical report [18]. This means that it is impossible to carry through all possible optimizations by hand. An implementation of the developed theory is necessary for that (implementing the theory is discussed in section 11.6). However, it is easy to discover some conspicuous possibilities for optimizations and to perform them manually.

The structuring of the rest of the chapter is as follows. Section 10.2 presents an introduction in the case specification and section 10.3 an introduction in the specification language ASF+SDF. In section 10.4 for each of the optimizations of chapter 9 some instances will be traced and be dealt with in the case specification. The presentation of section 10.4 parallels the presentation of the various optimizations of chapter 9; there will be a subsection on rule reductions, one on size reductions, one on vocabulary reductions, and one on the acquiring of special properties.

10.2 Static Semantics of Pascal

The case of this chapter is based on the algebraic specification for the static semantics of Pascal by Arie van Deursen, as documented in [18]. By "static semantics of a Pascal program" is meant "all properties of the program that can be checked without executing it" [18]. The specification actually comprises the syntax rules of the Pascal programming language [35] and, based on this, functions for checking the static semantics. The syntax rules correspond essentially to an equation free many sorted signature and the semantic rules to an additional many sorted signature part and equations (rewrite rules in our terminology) to define the semantics checking functions. The bulk of the specification is for the definition of these semantics checking functions; the syntax rules for Pascal require only 6 pages. It turns out that most of the semantic properties to be checked deal with the type and scope rules of Pascal.

Despite the problems mentioned in the introduction this case is well suitable for our purpose. The specification is put in such a way that opportunities for optimizations are abundant. This is because readability was the highest priority issue in the design phase. In fact the theory of this dissertation is a direct answer to a
question that Arie van Deursen raises in his technical report. A quotation from this report:

"Another question concerns the style of the specification. Inspired by the objective of readability, a software engineering specification style has been used, resulting sometimes in verbose names or superfluous functions. It would also be interesting to construct a specification using a more mathematically oriented style, e.g., with short names and the absolute minimum number of functions" [18].

The theory of this dissertation enables us to convert a specification with a software engineering style into one with a mathematically oriented style automatically. Peculiar enough the quotation does not mention the most important differences between a software engineering specification style and a mathematically oriented style, in our opinion. The length and verbosity of names is a mere syntactical and superficial aspect which has nothing to do with the structure of the specification. Instead, besides the number of functions (constructors in our terminology), the number of rules is an important aspect of the specification style. We will show that both the number of functions (constructors) and the number of rules can be reduced, thus attaining a "mathematically oriented specification style".

Some short remarks about the contents of the specification will be given here. For a more detailed description one is referred to the original report [18]. The specification is divided into five groups of modules (modularization of ASF+SDF specifications will be addressed in section 10.3). The subjects of these groups are:

1. Syntax,
2. Miscellaneous,
3. Contexts,
4. Environments,
5. Type checking.

Each group consists of a number of ASF+SDF modules, 6-12 modules per group. Some modules, like the basic module for boolean values and operations, appear in multiple groups. A short description of the functionality of each group is as follows.

Syntax. In this group the lexical and context free syntax of Pascal programs is defined. It is actually the original BNF-grammar of Pascal, as given in [35], with slight modifications, because ASF+SDF offers more powerful constructions for disambiguation (section 10.3).

Miscellaneous. In this group some necessary basic notions such as booleans, numbers, and integers with arithmetic, are defined.

Context. In this group the context notion is defined formally. A context is a compact representation of all definitions and declarations which are visible at a certain point in a Pascal program. A context is essentially a stack-like data structure which can hold entries that represent constant definitions, type definitions, variable
declarations, etc. Upon reaching a new "block" in a Pascal program, a new frame is pushed onto this stack, and upon leaving the block this frame is popped from the stack. The frame consists of a special block-mark entry, followed by a number of definition entries.

*Environments.* In this group the environment notion is defined formally. The environment is a data-structure for the communication between the various type checking functions, to be defined in the next group. An environment consists of a sequence of values, contexts, and error messages.

*Type checking.* In this group a number of functions are defined formally that perform various semantic checks on syntactically correct Pascal programs. The functions communicate via environments and the final result of the "main" type check function applied to an entire program, is a list of error messages, possibly (preferably) empty.

There is a fundamental difference in the way syntactic and static-semantic errors in a Pascal program are reported. Syntax errors will emerge in the implemented ASF+SDF system, whenever it is not able to parse the given program according to the specified grammar. Static-semantic errors will emerge in the system that is specified in the ASF+SDF language.

Being aware that this introduction in the case specification is very brief, we refer the reader to the original technical report [18] for more details.

### 10.3 The Specification Language

A concise introduction into the ASF+SDF language and system [30] will be given in this section. ASF+SDF is the combination of two formalisms which were developed separately. ASF, an acronym for Algebraic Specification Formalism [6, 29], is a many sorted algebraic specification formalism supporting modularization and conditional equations. The ASF system was able to generate concrete implementations of rewrite systems from specifications, and to execute them. SDF, an acronym for Syntax Definition Formalism [28], supports the definition of lexical, context-free, and abstract syntax at the same time. Incremental parsers can be generated from these definitions. The combination of ASF and SDF is a powerful tool that pairs the benefits of many sorted algebraic specifications (proof power, guaranteed free of ambiguities) with various kinds of syntactic freedom.

An interesting aspect of the SDF formalism is the relation it defines between context free grammars and many sorted signatures. An SDF specification can be seen as a context free grammar specification by just reading the lines of the specification in the opposite direction [28]. The gap between sorted signatures and unambiguous context free grammars is narrowed by offering a great deal of syntactic freedom with the specification of the function names (*constructors* in our terminology) and by offering powerful disambiguation constructions like *priority* and *associativity* declarations. These constructions can be used to disambiguate terms which can otherwise be parsed in several ways. An example will be given of the relation between context
free grammars and sorted signatures in ASF+SDF and one of the use of disambiguation primitives. The syntax of ASF+SDF specifications themselves is easy to understand and will be presented more detailed later.

Example 10.1. Consider the following part of an ASF+SDF specification for an illustration of the close relation that ASF+SDF defines between many sorted signatures and context free grammars.

```
exports
sorts ELEMENT PAIR
context-free syntax
    "this" ELEMENT "is associated with that" ELEMENT -> PAIR
```

This is a fragment of a specification of a function "is-associated-with" that maps two elements (of sort ELEMENT) to one pair (of sort PAIR). A more conventional notation for this specification would be:

```
is-associated-with : ELEMENT x ELEMENT -> PAIR.
```

The syntactic freedom that is offered by SDF brings the specification closer to context free grammars. In the ASF+SDF syntax rule the quoted parts can be seen as a "mixed-fix" operator. If the syntax rule of this specification is read as:

```
PAIR ::= "this" ELEMENT "is associated with that" ELEMENT
```

we have a production of a context free grammar in BNF [2], in which PAIR and ELEMENT are non-terminals and "this" and "is associated with that" terminals (definition 5.44).

Example 10.2. Consider the following part of an ASF+SDF specification for an illustration of the use of disambiguation primitives.

```
exports
sorts BOOL
context-free syntax
    "F" -> BOOL
    "T" -> BOOL
    BOOL "and" BOOL -> BOOL {left}
    BOOL "or" BOOL -> BOOL {left}
priorities
    "or" < "and"
```
Here "and" has a higher priority than "or", thus stating that, for instance, the term: T or F and F should be interpreted as: T or (F and F) rather than as: (T or F) and F. Moreover, both "and" and "or" are defined to be left associative, that is: T and T and T is interpreted as: (T and T) and T rather than as: T and (T and T).

An ASF+SDF specification consists of one or more modules. The modularization of algebraic specification is treated in detail in [25]. A module in the ASF+SDF formalism may contain distinguished sections which are prefaced by the following keywords:

- imports, listing the names of imported modules;
- exports, listing the exported or hidden items (sorts and functions);
- hiddens, listing the hidden (encapsulated) items;
- sorts, listing the sorts;
- lexical syntax, giving lexical syntax;
- context-free syntax, listing the function declarations. Attributes, between \{\} brackets, may be associated with functions, stating for instance that the function is left or right associative;
- priorities, giving priorities between functions;
- variables, declaring the variables used in the equations;
- equations, listing the (conditional) equations of the module.

The ASF+SDF specification has to be converted to our retrenched vocabulary formalism before it can be used to demonstrate our theory. Because ASF+SDF context-free syntax rules most closely resemble context free grammars (the syntax rules are actually grammar productions in reversed notation), the remarks about the conversion of context free grammars into vocabularies (section 5.4) apply. This means that each context-free syntax rule of the specification has to be associated with one unique constructor (section 5.5, example 5.45). This is self evident for syntax rules that have a unique terminal or a unique combination of terminals in the left hand part like for instance the syntax definition of a Pascal choice statement:

"if" EXPR "then" STAT "else" STAT -> STAT,

which will be transformed into:

\[ if\text{-}then\text{-}else: \text{EXPR} \times \text{STAT} \times \text{STAT} \to \text{STAT}. \]
10.4. THE OPTIMIZATIONS

However, it is less obvious for syntax rules like for instance the syntax definition of the Pascal "block"-structure in terms of definitions, declarations, and a compound statement:

```
DEFS DECLS COMP-STAT -> BLOCK,
```

In this case a function:

`block-constructor: DEFS × DECLS × COMP-STAT → BLOCK`

has to be introduced. Care has to be taken in situations in which different syntax rules have identical terminal strings or identical combinations of terminal strings.

Notational conventions are as follows. ASF+SDF specifications (or fragments of ASF+SDF specifications) are printed with a straight typewriter font and vocabulary specifications are printed with a slanted (italic) font. Moreover:

- constants and function (constructor) names are printed in lowercase,
- variable names are printed starting with an uppercase and the remaining part lowercase,
- sort (type) names are printed entirely in uppercase (single uppercase letters always denote variables).

The original text of the case specification does not conform entirely to this convention. Notation is therefore slightly adjusted at some places.

The notation for constants and functions in ASF+SDF (literals, or terminals in the grammar interpretation of ASF+SDF specifications) has to be quoted if the strings are keywords in the ASF+SDF language. Quoting of literals is always possible; one is encouraged to use quotation marks whenever this improves readability.

10.4 The Optimizations

10.4.1 Introduction

In this section parts of the entire specification will be used to demonstrate the various optimization techniques of chapter 9. For each optimization a part of the specification will be isolated, converted into the vocabulary formalism, and the optimization will be carried through. Section 10.4.2 deals with rule reduction, section 10.4.3 with size reductions, section 10.4.4 with vocabulary reductions, and section 10.4.5 finally deals with acquiring special properties.
10.4.2 Rule Reductions

The opportunities for rule reductions in the specification are abundant. We will discuss a number of them.

Example 10.3. Consider the following rules that are part of the module context-diffs.

[d11] C defines Id as a function? =
     C defines Id as a normal function? | C defines Id as a predefined function?
[d12] C defines Id as a procedure? =
     C defines Id as a normal procedure? | C defines Id as a predefined procedure?

which is, after conversion to the vocabulary formalism:

[d11] \( \text{defines-as-a-function}(C, \text{Id}) \Rightarrow \)
     or(\(\text{defines-as-normal-function}(C, \text{Id}),\)
     \(\text{defines-as-predefined-function}(C, \text{Id})\))
[d12] \( \text{defines-as-a-procedure}(C, \text{Id}) \Rightarrow \)
     or(\(\text{defines-as-normal-procedure}(C, \text{Id}),\)
     \(\text{defines-as-predefined-procedure}(C, \text{Id})\)).

The two rules define two functions: \(\text{defines-as-function}\) and \(\text{defines-as-procedure}\), which check if a certain given identifier, to be bound to the variable \(\text{Id}\), is defined as a (Pascal) function or procedure in a certain context, to be bound to the variable \(C\). A context is a compact representation of all definitions and declarations that are visible at a certain point of the program. The two functions can be seen as predicates, since they return boolean values. The function \(or\) (binary infix operator "|" in the ASF+SDF text) is defined elsewhere in the specification (module \text{Booleans}).

The two rewrite rules strongly suggest that somehow a generalization is possible. This is merely because of the choice of the names of the constructors. All constructors that start with "defines" are actually distinct and a lot has to be done before the conditions for a generalization do hold.

Intuitively: the function and procedure notions have to be represented by a more general "function-or-procedure" notion which can be specialized, instantiated, by a constant "\text{func}" or "\text{proc}". The rules with generalized constructors are:

[d11] \( \text{defines-as-funcproc}(\text{func}, C, \text{Id}) \Rightarrow \)
     or(\(\text{defines-as-normal-funcproc}(\text{func}, C, \text{Id}),\)
     \(\text{defines-as-predefined-funcproc}(\text{func}, C, \text{Id})\))
[d12] \( \text{defines-as-funcproc}(\text{proc}, C, \text{Id}) \Rightarrow \)
     or(\(\text{defines-as-normal-funcproc}(\text{proc}, C, \text{Id}),\)}
defines-as-predefined-funcproc(proc, C, Id)).

Now the two rules can be generalized over the constants func and proc, resulting in the single rule:

\[
\text{defines-as-funcproc}(V, C, Id) \Rightarrow \\
\quad \text{or}(\text{defines-as-normal-funcproc}(V, C, Id), \\
\quad \text{defines-as-predefined-funcproc}(V, C, Id)),
\]

with \( V \) a variable that ranges over the set \{func, proc\}. Note that this variable is maximally typed (definition 2.21) because there is no reason to define the permission (the \( \rho \)-value) of the first operand positions of the three generalized predicates any larger than \{func, proc\}.

The generalizations would have been found by the strategy of section 9.3, but it is clearly a non-trivial case. For a generalization to be performed on the initial rules the constructors defined-as-function has to be identified with defined-as-procedure, defined-as-normal-function with defined-as-normal-procedure, and defined-as-predefined-function with defined-as-predefined-procedure (section 9.3.2). This generalization becomes possible only after an expansion of these constructors into new ones with fixed constants at the new operand positions. Initially the six new fixed constants are all distinct, but they can be identified into the two constants func and proc. After this identification the identification of the predicates is possible and the conditions for a generalization can be made to hold. Each of the separate steps is among the “techniques” of section 9.3, so the final situation is a “white leaf” in the decision tree of the strategy of section 9.3 and descending sufficiently deep in this tree would have revealed this leaf.

It is easy to see that the transformations that enable this generalization do not have any multiplying effects on other rules in which the predicate functions occur since each occurrence of the original predicates can be constructed by instantiating the corresponding generalized predicate. However, the transformations do affect the size of the terms because terms and rules in which the original predicates occur will receive one or more additional constants.

\[\square\]

**Example 10.4.** The following five equations from module Context-search allow a very easy generalization. An excerpt from Context-search:

\[
\begin{align*}
[pd1] \text{integer-type} & = \text{find integer in init-block} \\
[pd2] \text{boolean-type} & = \text{find boolean in init-block} \\
[pd3] \text{char-type} & = \text{find char in init-block} \\
[pd4] \text{text-type} & = \text{find text in init-block} \\
[pd5] \text{real-type} & = \text{find real in init-block}
\end{align*}
\]

In order to convert these equations to the vocabulary formalism one has to know that integer-type, boolean-type, etc. are constants, integer, boolean,
etc. are constants, init-block is a constant, and that find...in... is a binary function with signature:

\[ \text{find SYMBOL in CONTEXT} \rightarrow \text{CONTEXT}. \]

After the conversion the rules are:

\[ \begin{align*}
& \text{pd1]} \quad \text{integer-type} \Rightarrow \text{find-in-context(}\text{integer, init-block}), \\
& \text{pd2]} \quad \text{boolean-type} \Rightarrow \text{find-in-context(}\text{boolean, init-block}), \\
& \text{pd3]} \quad \text{char-type} \Rightarrow \text{find-in-context(}\text{char, init-block}), \\
& \text{pd4]} \quad \text{text-type} \Rightarrow \text{find-in-context(}\text{text, init-block}), \\
& \text{pd5]} \quad \text{real-type} \Rightarrow \text{find-in-context(}\text{real, init-block}). \\
\end{align*} \]

The rules define the meaning of certain constants in terms of the earlier defined entities init-block and integer, boolean, etc. A generalization over these rules is now very easy by generalizing the constants integer-type, boolean-type, etc. This is possible by creating one unary constructor which can be instantiated by the integer, boolean, etc. constants. The new constructor will be named ibctr-type (a concatenation of the first letters of the involved types). The rules become:

\[ \begin{align*}
& \text{pd1]} \quad \text{ibctr-type(}\text{integer}) \Rightarrow \text{find-in-context(}\text{integer, init-block}), \\
& \text{pd2]} \quad \text{ibctr-type(}\text{boolean}) \Rightarrow \text{find-in-context(}\text{boolean, init-block}), \\
& \text{pd3]} \quad \text{ibctr-type(}\text{char}) \Rightarrow \text{find-in-context(}\text{char, init-block}), \\
& \text{pd4]} \quad \text{ibctr-type(}\text{text}) \Rightarrow \text{find-in-context(}\text{text, init-block}), \\
& \text{pd5]} \quad \text{ibctr-type(}\text{real}) \Rightarrow \text{find-in-context(}\text{real, init-block}). \\
\end{align*} \]

Now the five rules can be generalized over the constants integer, boolean, char, text, real, resulting in the single rule:

\[ \text{ibctr-type(}\text{V}) \Rightarrow \text{find-in-context(}\text{V, init-block}), \]

with V a variable that ranges over the set \{integer, boolean, char, text, real\}. Note that this variable is maximally typed because there is no reason to define the permission (the p-value) of the operand position of ibctr-type any larger than \{integer, boolean, char, text, real\}.

\[\Box\]

**Example 10.5.** The following equations from the module Context-defs are also interesting to demonstrate the possibilities for rule reductions. An excerpt from Context-defs:

\[ \begin{align*}
& \text{de2]} \quad \text{[Prefix, const Id = Const] defines Id as a constant? = true} \\
& \text{de3]} \quad \text{[Prefix, type Id = Type-den] defines Id as a type? = true} \\
& \text{de4]} \quad \text{[Prefix, var Id : Type-den] defines Id as a variable? = true} \\
\end{align*} \]
In the conversion to the vocabulary formalism the list primitives have to be eliminated. This is possible by means of the standard technique with a binary list constructor and a constant for the representation of the empty list.

\[
\begin{align*}
[de2] & \text{ defines-as-const}(\text{context-constr}(\text{Prefix}, \text{const-def}(\text{Id}, \text{Const})), \text{Id}) \\
& \Rightarrow \text{true}, \\
[de3] & \text{ defines-as-type}(\text{context-constr}(\text{Prefix}, \text{type-def}(\text{Id}, \text{Type-den})), \text{Id}) \\
& \Rightarrow \text{true}, \\
[de4] & \text{ defines-as-var}(\text{context-constr}(\text{Prefix}, \text{var-def}(\text{Id}, \text{Type-den})), \text{Id}) \\
& \Rightarrow \text{true}.
\end{align*}
\]

The rules define functions which check if a given identifier is defined as constant, type, or variable in the given context (actually in the last entry of that context). The last two rules can easily be generalized with the same technique as in example 10.4: generalizing the constructors \text{defines-as-type} and \text{defines-as-var} on the one hand and the constructors \text{type-def} and \text{var-def} on the other hand, and instantiating these generalizations with the constants \text{type} and \text{var}. The generalized rule is:

\[
\text{defines-as-tv}(V, \text{context-constr}(\text{Prefix}, \text{tv-def}(V, \text{Id}, \text{Type-den})), \text{Id}) \\
\Rightarrow \text{true},
\]

with \(V\) a variable that ranges over the set \{\text{type}, \text{var}\}.

It is possible to add the first rule to this generalization but one has to resort to un-elegant tricks for that. Consider the second operand of \text{const-def}. This operand, a variable, has a type which is different from the types of the second operands of \text{type-def} and \text{var-def}. Because \text{const-def} has to be typed correctly, a generalization of \text{const-def}, \text{type-def}, and \text{var-def} cannot be instantiated with a constant. The second operand has a type which is only known after instantiation, so the only possibility is to situate the second operand position below the instantiating constructor, which has to be an unary constructor therefore. The generalization of the \text{defines-as} constructors has to be instantiated with the same unary constructor. The operand position of this constructor now has to be filled with something: a dummy. Because of the typing this has to be a dummy constant or a dummy type denoter. Concrete: generalizing the three \text{defines-as} and the three \text{def} constructors leads to the rules:

\[
\begin{align*}
[de2] & \text{ defines-as-ctv}(\text{const}(0), \\
& \text{context-constr}(\text{Prefix}, \text{ctv-def}(\text{Id}, \text{const(\text{Const}))}), \text{Id}) \\
& \Rightarrow \text{true}, \\
[de3] & \text{ defines-as-ctv}(\text{type}(\text{char}), \\
& \text{context-constr}(\text{Prefix}, \text{ctv-def}(\text{Id}, \text{type(\text{Type-den}))}), \text{Id}) \\
& \Rightarrow \text{true}, \\
[de4] & \text{ defines-as-ctv}(\text{var}(\text{char}), \\
& \text{context-constr}(\text{Prefix}, \text{ctv-def}(\text{Id}, \text{var(\text{Type-den}))}), \text{Id}) \\
& \Rightarrow \text{true}.
\end{align*}
\]
At this stage the three rules can be generalized to:

\[
\text{defines-as-ctv}(V, \text{context-constr}(\text{Prefix, ctu-def}(\text{Id}, V)), \text{Id}) \Rightarrow \text{true}.
\]

with \( V \) a variable that ranges over the set \{const, type, var\}. \( \square \)

**Example 10.6.** In the following four rules from module Context-sens-funs an easy generalization is possible in the last two rules. The first two rules resemble each other very much, no generalization is possible however. It will be discussed why. An excerpt from module Context-sens-funs:

\[
[\text{ac1}] \quad \text{[Prefix, type Id = Ps array [Index-type] of Comp-type].} \\
\quad \text{index-type = get-type of Index-type in [Prefix]} \\
[\text{ac2}] \quad \text{[Prefix, type Id = Ps array [Index-type] of Comp-type].} \\
\quad \text{comp-type = get-type of Comp-type in [Prefix]} \\
[\text{ac3}] \quad \text{[Prefix, type Id = Ps set of Comp-type].comp-type =} \\
\quad \text{get-type of Comp-type in [Prefix]} \\
[\text{ac3}] \quad \text{[Prefix, type Id = Ps file of Comp-type].comp-type =} \\
\quad \text{get-type of Comp-type in [Prefix]}
\]

The specification rules need a little explanation. The outermost square brackets of the rules are part of the ASF+SDF syntax, whereas the innermost square brackets of ac1 and ac2 are part of the Pascal syntax. The first equal sign in each rule (after the \text{Id} variable) is part of the Pascal syntax, whereas the second equal sign in each rule is part of the ASF+SDF syntax (it separates the left hand side from the right hand side of the transformation rule, which is called "equation" in ASF+SDF). The variable Ps in each rule represents the fact whether or not the defined type is a packed type (a "packed" type is implemented in a memory efficient way [35]).

In the conversion to the vocabulary formalism the sequence primitives have to be simulated in the standard way with a binary list constructor and a constant representing the empty list. After this conversion the rules are:

\[
[\text{ac1}] \quad \text{get-index-type(context-constr(\text{Prefix,} \\
\quad \text{array-constr(Ps, Index-type, Comp-type))})} \\
\quad \Rightarrow \text{get-type(\text{Prefix, Index-type}),} \\
[\text{ac2}] \quad \text{get-comp-type(context-constr(\text{Prefix,} \\
\quad \text{array-constr(Ps, Index-type, Comp-type))})} \\
\quad \Rightarrow \text{get-type(\text{Prefix, Comp-type}),} \\
[\text{ac3}] \quad \text{get-comp-type(context-constr(\text{Prefix,} \\
\quad \text{set-constr(\text{Comp-type}))})} \\
\quad \Rightarrow \text{get-type(\text{Prefix, Comp-type})} \\
[\text{ac4}] \quad \text{get-comp-type(context-constr(\text{Prefix,} \\
\quad \text{file-constr(\text{Comp-type}))})}
\]
10.4. THE OPTIMIZATIONS

⇒ get-type(Prefix, Comp-type)

The rules define access functions for the component types of structured types: index types and component types of array types and component types of set types and file types. The last two rules can be generalized in the following way:

get-comp-type(context-constr(Prefix, sf-constr(V, Comp-type)))
⇒ get-type(Prefix, Comp-type)

with V a variable that ranges over the set of constants \{set, file\}. The first two rules cannot be generalized, although they look very similar. The difference between index type and component type of an array type means in practice the choice between the two variables Index-type and Comp-type. There is no way of doing this in one single rule.

10.4.3 Size Reductions

Unlike rule reductions there is no absolute limitation on the reduction of sizes of terms (as long as they are not already collapsed into one single constant). The only limiting factor in size reductions may be the fact that size reductions sometimes have an increase in the number of rules or the size of the vocabulary as a consequence. We will therefore only consider size reductions which are not involved in this kind of trade-offs. A number of examples will be presented, which are all based on the syntax definition rules of Pascal statements. The module SyntaxStats is defined as follows in the case specification:

module SyntaxStats

imports SyntaxExpr

exports

sorts

UNL-STAT SIMPLE-STAT STRUCT-STAT STATEMENT
CASE-ELT COMP-STAT DOWN-TO

context-free syntax

LABEL ":" UNL-STAT
⇒ STATEMENT
UNL-STAT
⇒ STATEMENT

SIMPLE-STAT
⇒ UNL-STAT
STRUCT-STAT
⇒ UNL-STAT
VAR-ACCESS ":=" EXPRESSION \rightarrow \texttt{SIMPLE-STAT}\\
IDENT \texttt{ACTUAL-PAR-LIST} \rightarrow \texttt{SIMPLE-STAT}\\
goto \texttt{LABEL} \rightarrow \texttt{SIMPLE-STAT}\\

\texttt{COMP-STAT} \rightarrow \texttt{STRUCT-STAT}\\
\texttt{begin \{STATEMENT ";\}+ end} \rightarrow \texttt{COMP-STAT}\\
\texttt{if EXPRESSION then STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\
\texttt{if EXPRESSION then STATEMENT else STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\
case EXPRESSION of \{\texttt{CASE-ELT ";\}+}\\
\texttt{OPT-SEMI-COLON end} \rightarrow \texttt{STRUCT-STAT}\\
\{\texttt{CONST ";\}+ ":\} STATEMENT \rightarrow \texttt{CASE-LIST-ELT}\\
\texttt{repeat \{STATEMENT ";\}+ until EXPRESSION} \rightarrow \texttt{STRUCT-STAT}\\
\texttt{while EXPRESSION do STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\
\"to\" \rightarrow \texttt{DOWN-TO}\\
downto \rightarrow \texttt{DOWN-TO}\\

\texttt{for IDENT ":=" EXPRESSION DOWN-TO EXPRESSION}\\
\texttt{do STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\
\texttt{with \{VAR-ACCESS ";\}+}\\
\texttt{do STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\

\texttt{priorities}\\
\texttt{if EXPRESSION then STATEMENT else STATEMENT} \rightarrow \texttt{STRUCT-STAT}\\
> \rightarrow \texttt{STRUCT-STAT}\\

\texttt{end-module SyntaxStats}

In the conversion to the ordinary many sorted signature formalism the sequence primitives have to be implemented with binary list constructors and constants representing empty lists. We will not go beyond a representation in the many sorted signature formalism. Readability will not benefit from a representation in the entrenched vocabulary formalism. The shape of the \(\rho\)-functions follows immediately from matches between operand types and result types of operators.

\texttt{label-constr: LABEL \times UNL-STAT} \rightarrow \texttt{STATEMENT}\\
\texttt{non-label-constr: UNL-STAT} \rightarrow \texttt{STATEMENT}\\

\texttt{simple-stat-constr: SIMPLE-STAT} \rightarrow \texttt{UNL-STAT}\\
\texttt{struct-stat-constr: STRUCT-STAT} \rightarrow \texttt{UNL-STAT}\\

\texttt{assignment: VAR-ACCESS \times EXPRESSION} \rightarrow \texttt{SIMPLE-STAT}\\
\texttt{procedure-call: IDENT \times \texttt{ACTUAL-PAR-LIST}} \rightarrow \texttt{SIMPLE-STAT}
10.4. THE OPTIMIZATIONS

\[
goto: \text{LABEL} \quad \rightarrow \text{SIMPLE-STAT} \\
\text{comp-stat-constr: COMP-STAT} \quad \rightarrow \text{STRUCT-STAT} \\
\text{begin-end: STAT}^+ \quad \rightarrow \text{COMP-STAT} \\
\text{stat}^+-\text{constr: STAT} \times \text{STAT}^* \quad \rightarrow \text{STAT}^+ \\
\text{stat}^-\text{constr: STAT} \times \text{STAT}^* \quad \rightarrow \text{STAT}^* \\
\text{empty-stat:}. \quad \rightarrow \text{STAT}^* \\
\text{if-then: EXPR} \times \text{STATEMENT} \quad \rightarrow \text{STRUCT-STAT} \\
\text{if-then-else: EXPR} \times \text{STATEMENT} \times \text{STATEMENT} \quad \rightarrow \text{STRUCT-STAT} \\
\text{case-constr: EXPR} \times \text{CASE-ELT}^+ \times \text{OPT-SEMI-COLON} \quad \rightarrow \text{STRUCT-STAT} \\
\text{case}^+\text{-constr: CASE-ELT} \times \text{CASE-ELT}^* \quad \rightarrow \text{CASE-ELT}^+ \\
\text{case}\text{-constr: CASE-ELT} \times \text{CASE-ELT}^* \quad \rightarrow \text{CASE-ELT}^* \\
\text{empty-case:} \quad \rightarrow \text{CASE-ELT}^* \\
\text{case-elt-constr: CONSTANTS} \times \text{STATEMENT} \quad \rightarrow \text{CASE-ELT} \\
\text{const}^+\text{-constr: CONSTANTS} \times \text{CONSTANTS} \quad \rightarrow \text{CONSTANTS}^+ \\
\text{const}^-\text{-constr: CONSTANTS} \times \text{CONSTANTS} \quad \rightarrow \text{CONSTANTS}^* \\
\text{empty-const:} \quad \rightarrow \text{CONSTANTS}^* \\
\text{repeat-until: STAT}^+ \times \text{EXPR} \quad \rightarrow \text{STRUCT-STAT} \\
\text{while-do: EXPR} \times \text{STATEMENT} \quad \rightarrow \text{STRUCT-STAT} \\
\text{to:} \quad \rightarrow \text{DOWN-TO} \\
\text{down-to:} \quad \rightarrow \text{DOWN-TO} \\
\text{for: IDENT} \times \text{EXPR} \times \text{DOWN-TO} \times \text{EXPR} \times \text{STATEMENT} \quad \rightarrow \text{STRUCT-STAT} \\
\text{with: VAR-ACCESS}^+ \times \text{STATEMENT} \quad \rightarrow \text{STRUCT-STAT}
\]

Potential candidates for size reductions without unwanted consequences for the number of rules and the number of constructors in the vocabulary are contractions associated with constants and unary constructors in the vocabulary. The five examples to come are all based on this kind of contractions.

**Example 10.7.** An almost trivial instance of size reduction is the incorporation of the constants “to” and “down-to” with type DOWN-TO in the constructor “for”. Examination of the entire specification reveals that the type DOWN-TO does not occur anywhere else than in the signature of the constructors to, down-to, and for. Moreover, the constants to and down-to do not occur in any rewrite rule. This means that the contraction of for with to, and for with down-to does not increase rule numbers and will actually decrease constructor numbers. The signature specifications:
to: \[ \rightarrow DOWN-TO \]
\[ \rightarrow DOWN-TO \]
\[ \text{for: } IDENT \times EXPR \times DOWN-TO \times EXPR \times \text{STATEMENT} \]
\[ \rightarrow STRUCT-STAT \]
can be replaced by:
\[ \text{for-to: } IDENT \times EXPR \times EXPR \times \text{STATEMENT} \]
\[ \rightarrow STRUCT-STAT \]
\[ \text{for-down: } IDENT \times EXPR \times EXPR \times \text{STATEMENT} \]
\[ \rightarrow STRUCT-STAT \]

\[ \square \]

**Example 10.8.** Another contraction without consequences for the number of rules and only a decreasing effect on the number of constructors is the contraction of the constructors *comp-stat-constr* and *begin-end*. The presence of the constructor *comp-stat-constr* seems to have no other purpose than to enable a clear distinction between the module *SyntaxStats* for the syntax of statements and the module *SyntaxProgram* for the syntax of entire programs. The Pascal compound statement which is defined by the syntax rule:

\[
\text{begin } \{\text{STATEMENT }";"\}^+ \text{ end } \rightarrow \text{COMP-STAT}
\]

is an ordinary statement, which is allowed at any place where any other statement is allowed. However, it is the only statement allowed as body of a program. The desire not to discriminate between the body of a program and a compound statement as ordinary statement necessitates the presence of the *COMP-STAT* type, the *comp-stat-constr* constructor, and the in other respects useless chain:

\[
\text{COMP-STAT } \rightarrow \text{STRUCT-STAT}
\]

The constructors *comp-stat-constr* and *begin-end* can be contracted without preliminary measures. In that case the property of being normalized (definition 5.12) will be lost and the specification will become inherently order sorted (theorem 5.42). The reason is that the constructor which is the result of the contraction, we will call it *contracted-comp-stat-const*, is allowed below the "program constructor" (which is specified in the module *SyntaxProgram*) and allowed below various other constructors below which general statements can occur. It is easy to see that this
corresponds to properly included ρ-values (one ρ-value being \{contracted-comp-stat-constr\} and other ρ-values containing some other constructors besides \{contracted-comp-stat-constr\}). According to theorem 5.42 this means that the signature has become inherently order sorted.

Order-sortedness can be prevented by first performing a root side distinguishing transformation (section 6.5) on \textit{comp-stat-constr} such that the identity of this constructor is different when it occurs in a program block. After this distinguishing transformation the contraction of the specialized \textit{comp-stat-constr} and \textit{begin-end} can be done without introducing properly including overlap amongst ρ-values.

\begin{itemize}
\item \textbf{Example 10.9.} It appears that the constructors \textit{simple-stat-constr} and \textit{struct-stat-constr} serve no operational purpose at all. The presence of these constructors is a remnant of a categorization of statements into "simple statements" and "structure statements" from the original Pascal specification [35]. This distinction is not necessary because simple statements are allowed to occur anywhere structure statements are allowed, and vice versa. This means that \textit{simple-stat-constr} and \textit{struct-stat-constr} can be identified (leaf side identification, section 6.7). After this identification, the pre-conditions of which are easily be seen to hold, the generalized constructor can be contracted with any constructor that is allowed below it (the statement constructors). The result is a reduction in size of terms and size of vocabulary because \textit{simple-stat-constr} and \textit{struct-stat-constr} have been eliminated completely without introducing new constructors (the new constructors that result from the contractions are essentially renamings of the original ones).
\end{itemize}

\begin{itemize}
\item Finally we will discuss some size reductions that do not affect rule numbers and vocabulary size but which will nullify the property of being normalized, a property which holds for converted ASF+SDF specifications because ASF is a many sorted algebraic specification formalism.
\end{itemize}

\begin{itemize}
\item \textbf{Example 10.10.} If many-sortedness in not to be preserved, the \textit{non-label-constr} constructor can be eliminated. The only reason for the presence of \textit{non-label-constr} and the type \textit{UNL-STAT} is to prevent statements from being labeled multiply. Labeled statements are allowed anywhere unlabeled statements are allowed, except after a label. The most natural solution for this situation is a solution with "properly including ρ-values", namely a set of constructors (below \textit{label-constr}) and this set of constructors extended with \textit{label-constr} (all positions where statements may occur). This solution is inherently order sorted (theorem 5.42). The elimination of the \textit{non-label-constr} can be achieved by first contracting \textit{non-label-constr} with each of the statement constructors that are allowed below it and after that identifying each contracted statement constructor with its original counterpart. E.g. the contraction leads to the new statement constructors (\textit{nonl} stands for \textit{non-labeled}):
\begin{itemize}
\item \textit{nonl-assignment},
\item \textit{nonl-procedure-call},
\end{itemize}
\end{itemize}
• nonl-goto,
• etc.

After these contractions nonl-assignment can be identified with assignment, nonl-procedure-call with procedure-call etc. It is easy to see that the pre-conditions for these identification hold: the set of constructors that can occur above the non-labeled constructors and the set of constructors that can occur above the original constructors \(\{\text{label-constr}\}\) are disjunct.

**Example 10.11.** The list\(+\)-constr constructors can be eliminated with similar consequences as in example 10.10: losing many-sortedness. Consider the following signature:

\[
\text{begin-end: STAT}^+ \rightarrow \text{COMP-STAT}.
\]

The type STAT\(^+\) is to prohibit empty statement lists in a compound statement. In the retrenched vocabulary formalism this objective can be met by allowing stat\(^+\)-constr below begin-end and prohibiting empty-stat below begin-end. However, this introduces proper inclusion amongst \(\rho\)-values because stat\(^+\)-constr and empty-stat are both allowed at the second position of stat\(^+\)-constr.

### 10.4.4 Vocabulary Reductions

Some of the size reductions of example 10.7 are also vocabulary reductions. This is at least the case with the elimination of the to and down-to constants and the elimination of the simple-stat-constr and struct-stat-constr constructors. Vocabulary reductions can be attained by means of contraction transformations and identifications (section 9.5). The elimination of to and down-to is an example of a vocabulary reducing contraction (sequence) and the elimination of simple-stat-constr and struct-stat-constr is an example of a vocabulary reducing identification.

### 10.4.5 Acquiring Special Properties

The case specification is not very suitable to illustrate the acquiring of special properties because the initial specification already possesses most of the properties. However, it is instructive to discuss the reason why the conditions of the properties do hold and with what transformations the properties get lost.

**Integratedness.** The case specification possesses the integratedness property (definition 5.9) but proving this is a non-trivial task, mainly because of the modularized representation of the specification. First the remark that the integratedness property always holds for vocabularies which are derived from formalisms that do not somehow support a primitive to restrict the number of constructors that can act as root of well-formed terms. In these formalisms any constructor can act as root of a term, so any constructor can occur in well-formed terms, and the condition for integratedness holds.
The ASF+SDF formalism does have a primitive to restrict the number of constructors that can act as root of well-formed terms but this primitive is not powerful enough for our intended purpose. The primitive is integrated in the ASF+SDF module concept by the imports, exports, and hidden notions (section 10.3). Sorts (types) that are exported are visible outside the module, whereas types that are specified as hidden are only for internal use. This means that only terms with a type which is exported from some module of the specification are well-formed with regard to the entire specification. The start set of allowed root constructors consists therefore of constructors with a result type which is exported from some module.

The imports, exports, and hidden primitives are not powerful enough to support both the modularization purposes and the desire to exclude certain constructors from occurring at root positions of well-formed terms, because ASF+SDF does not offer a primitive to specify which modules are "main" and which modules "auxiliary". In other words: it is not possible to exclude from occurring at root positions of terms constructors with a result type which is exported from an auxiliary module. The problem is apparent in the "syntax" group of the specification. In this group the Pascal syntax is defined with not less than eight modules. The modules carry self explanatory names like SyntaxProgram, SyntaxHeaders, SyntaxTypes, etc. It will be clear that the only real well-formed terms are terms with type PROGRAM. Allowing this the only type to be exported from a module means that there is no room left for modularization.

It is not difficult to verify that all constructors that are specified in some module of the case specification (exported as well as hidden constructors) can occur in a well-formed term with a type that is exported from some module, implying that the case specification is integrated with regard to its own (weak) primitives for the specification of a "root restriction set". The specification is in fact integrated with regard to the type ERRORS, or even stronger: with regard to the ERRORS typed constructor tc (module TC-program), the main type check function.

Non-integratedness will in fact only occur in intermediate stages after certain contraction or distinguishing transformations. The constants "to" and "down-to" in example 10.7 became non-integrated after the contractions with the for constructor. They simply have to be discarded to regain integratedness.

Satisfiability. Like with integratedness the satisfiability property holds for the initial case specification. The condition for this in terms of the many sorted ASF+SDF formalism is that there has to be at least one constructor with result type X for each type X that occurs as operand type of a constructor. The modularity concept causes some complications: there has to be at least one visible constructor with result type X (visible with regard to the exported, imported, and hidden concepts). It is not difficult to verify these conditions for the case specification.

Like with integratedness, the absence of the property, non-satisfiability, will usually only occur after a contraction transformation. Unsatisfiability will in particular occur after the contraction of a trivial combination (section 5.2). In that case the parent constructor of the contracted combination pa-ch (section 6.10) will
become non-satisfiable.

*Trivial combination freeness.* The initial case specification is not entirely free of trivial combinations. One trivial combination, the constructor pair `comp-stat-constr / begin-end` was eliminated in example 10.8. It is likely that there are more trivial combinations in the initial specification but they are hard to find without a proper tool. Note that the elimination of a trivial combination will always result in at least one non-satisfiable constructor and possibly also in one or more non-integrated constructors.

*Being normalized.* The property of being normalized, roughly equivalent to the property of being many sorted (as special case of order-sortedness) has already got some attention in previous sections. The initial case specification is normalized, for obvious reasons (theorem 5.42), and some transformations to reduce rule numbers, size of terms, or size of vocabulary will destroy the property (examples 10.10 and 10.11). It is in general possible to reduce rule numbers, term sizes, and vocabulary sizes further if the being normalized property does not need to be maintained. This is actually a way of saying that order sorted typing is more expressive than many sorted typing. This fact is generally known and documented amongst others in [54].

*Being uniquely starting.* A vocabulary (or signature) is uniquely starting if and only if the constructors which are allowed as root constructors of well-formed terms are not allowed to occur at internal positions of terms and vice versa (definition 5.13). The initial case specification is not uniquely starting. The constructor `program-constr` (module `SyntaxProgram`) is for instance allowed as root of well-formed terms (the type `PROGRAM` is exported from module `SyntaxProgram`), the constructor is also allowed below the `tc` constructor (module `TC-program`), the main type check function.

The absence of the uniquely starting property is related to the absence of powerful primitives to restrict the set of allowed root constructors in the ASF+SDF formalism. This was described already in the paragraph about integratedness. It is possible to transform the vocabulary representation of the initial specification into one which is uniquely starting but this is quite useless, since most of the constructors that are allowed both at root positions and at internal positions, do so because of the absence of a powerful root restriction primitive in the ASF+SDF formalism.

*Modular decomposability.* The modularity property that was mentioned in section 9.6.7 is quite different from the modularity concept as adopted in ASF+SDF. Although the case specification is stated in a modularized way (with regard to the ASF+SDF conception of modularity), we have not been able to discover modular subsets of rules, nor opportunities to create modular subsets of rules by syntactical transformations. The reason for this is mainly because of the absence of a proper tool to automatically process specifications in order to discover or to establish modular decomposability. The modularity concept as adopted for term rewriting systems [46], requires that the constructor subsets associated with distinct modules be disjunct. This is difficult to check for large specifications since each rule of a potential module has to be checked against the presence of constructors from the other module(s).

*Arity transformations.* Arity transformations are relatively simple transforma-
10.5. CONCLUSION

As stated in the introduction this case is a compromise in a number of respects. It is by no means an exhaustive example on all the techniques that are presented for each of the optimization criteria. Some techniques could not be illustrated by the lack of a good opportunity in the case specification, others could not be illustrated by the lack of an implementation of the optimization algorithms. Even with the optimization criterion which received most attention, the reduction of the number of rules, only a limited number of techniques for that optimization criterion have been demonstrated. We hope however that the case did demonstrate that practical specifications do offer opportunities for the application of the presented optimization techniques.
Chapter 11

Concluding Remarks

11.1 Introduction

In this chapter some suggestions for further research are given and some general remarks are presented, retrospectively, about the various design decisions of the presented formalisms, and about the way in which the research question is answered.

In section 11.2 the generalization of the theory for ground rewriting to general rewriting (of terms with variables) is discussed. The generalization of unconditional rewriting to conditional rewriting (rules with conditions) is subject of section 11.3. Section 11.4 summarizes some possibilities for further research at optimization strategies and section 11.5 is devoted to the implementation of the theory. In section 11.6 a justification for the introduction of the retrenched vocabulary formalism is presented and finally in section 11.7 the research question is reconsidered and its answering evaluated.

11.2 Generalizing Ground Rewriting

From chapter 6 and onwards the developed theory is restricted to ground rewriting, based on rules with variables. Practical applications of rewriting techniques are virtually always ground rewriting applications; chapter 10 contains examples. So the majority of practical cases is covered by the theory. The restriction to ground rewriting has the following reasons. Firstly: the theory for ground rewriting is in such a way more elegant and simpler that it will always remain desirable to have a separate formulation of the theory for this special case. Secondly: it is possible to generalize the theory to general rewriting but this requires a substantial amount of additional research effort. Ground rewriting is a special case of general rewriting and is much easier to deal with.

To be more concrete, the problems with general rewriting are as follows. Variables are a means to describe sets of terms in a generic way. If a term $t$ contains the variable $v$ then $t$ stands for the set of ground terms that result when substituting all possible different ground terms for $v$ (only type preserving substitutions). The
set of ground terms corresponding to one term with variables will be called \textit{generic term set} in the rest of this chapter. In general, terms with variables correspond to an infinite number of ground terms. Now the problem is that with some of the elementary transformations, a term set which can be represented by one single term (with variables) in the simulated domain, cannot always be represented by one single term in the simulating domain. An example will be presented to illustrate this.

**Example 11.1.** The lack of one-to-one correspondence between terms with variables in two different term domains. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

- \( C = \{ f, g, h, a, b \} \),
- \( V = \{ v, \ldots \} \),
- \( S = C \),
- \( \alpha(f) = 2, \alpha(g, h) = \{ 1 \}, \alpha(a, b) = \{ 0 \} \),
- \( \rho(f, 1) = \rho(f, 2) = \{ g, h, a, b \}, \rho(g, 1) = \rho(h, 1) = \{ f, g, h, a, b \} \),
- \( \mu(v) = \{ g, h, a, b \} \).

The term \( f(v, a) \) stands for the infinite set of terms:

\[ \{ f(a, a), f(b, a), f(g(a), a), f(g(b), a), f(h(a), a), f(h(b), a), f(g(f(a)), a), \ldots \}. \]

Leaf side distinguishing \( f \) with partition \( \{ \{ g, a \}, \{ h, b \} \} \) leads to a modified vocabulary \( A' = \langle C', V', S', \alpha', \rho', \mu' \rangle \) with:

- \( C' = \{ f_1, f_2, g, h, a, b \} \),
- \( \alpha'(f_1) = \alpha''(f_2) = \alpha(f) \),
- \( \rho'(f_1, 1) = \{ g, a \} \),
- \( \rho'(f_2, 1) = \{ h, b \} \),
- \( \rho'(f_1, 2) = \rho'(f_2, 2) = \{ g, h, a, b \} \),

the rest of \( A \) remains unchanged. Now the generic term set:

\[ \{ f_1(a, a), f_2(b, a), f_1(g(a), a), f_1(g(b), a), f_2(h(a), a), f_2(h(b), a), f_1(g(f(a)), a), \ldots \} \]
can no longer be described with one single term with variables. Two terms are necessary instead:

\[ \{f_1(v',a), f_2(v'',a)\}, \]

with \( \mu(v') = \{g,a\}, \mu(v'') = \{h,b\} \). Note that a single variable with permission \( \{f_1, f_2\} \) is not possible because this variable would include terms like \( f_1(a,b) \). The phenomenon of generic term sets which cannot be represented by one term with variables occurs with each of the distinguishing transformations and with contraction. A concrete example for the contraction case will be presented. Consider the vocabulary \( A = \langle C, V, S, \alpha, \rho, \mu \rangle \) with:

- \( C = \{f, g, a\} \),
- \( V = \{v, \ldots\} \),
- \( S = \{f\} \),
- \( \alpha(f) = 2, \alpha(g) = 1, \alpha(a) = 0 \),
- \( \rho(f, 1) = \rho(f, 2) = \{g, a\}, \rho(g, 1) = \{f, a\} \),
- \( \mu(v) = \{f, a\} \).

The term \( f(v, a) \) stands for the infinite set of ground terms:

\[ \{f(a, a), f(g(a), a), f(g(f(a, a)), a), f(g(f(g(a)), a), a), \ldots\} \]

Contracting \( f \) and \( g \) at position 1 of \( f \) into \( h \) leads to a modified vocabulary \( A' \) with:

- \( C' = \{f, g, h, a\} \),
- \( \alpha'(h) = 2 \),
- \( \rho'(f, 1) = \{a\} \),
- \( \rho'(h, 1) = \{f, h, a\} \),
- \( \rho'(h, 2) = \{g, a\} \),

the rest of \( A \) remains unchanged. Now the generic term set:

\[ \{f(a, a), h(a, a), h(f(a, a), a), h(h(a, a), a), \ldots\} \]

can no longer be described with one single term with variables. Two terms are necessary instead:

\[ \{f(a, a), h(v', a)\}, \]
with $\mu(v') = \{f, h, a\}$. A single variable with permission $\{f, h\}$ is not a solution
because this variable would include terms like $f(a, g(a))$. It is easy to see that
it is always possible to represent the generic term set of one term in the original
vocabulary with a finite number of terms in the new vocabulary.

Note also that the necessity to represent the generic term set of one term in
the original vocabulary with two or more terms in the new vocabulary does not
occur often in practice. If $f$ in the first and second part of the example is a unary
constructor then it is in both parts possible to represent the generic term sets with
one single term: in the first part with a variable $v$ with $\mu(v) = \{f_1, f_2\}$ and in the
second part with a variable $v$ with $\mu(v) = \{f, h\}$. A similar solution is possible in
the case of a binary constructor $f$ and a variable at the second position of $f$.  

The example suggests that a generalization of the theory for ground rewriting
to general rewriting can be attained by generalizing the simulation concept from
simulating single terms by single terms to simulating sets of terms by sets of terms.
One of the things that should be taken care of is the fact that one set of ground
terms can be represented in many different ways by sets of terms with variables. In
example 11.1 for instance, the generic term set of $\{f(v)\}$ is the same as the generic
term set of $\{f(v'), f(v'')\}$ that would be the result of an identification operation on
the vocabulary $A'$ of example 11.1 (first part). This suggests a kind of normal form
for sets of representatives. The development of this more general form of simulation,
with which it is possible to cover general rewriting, is a possible direction for further
research.

11.3 Adding Conditions to Rules

The addition of conditions to the rules of the formalism of term rewriting sys-
tems offers a substantial increase in expressiveness. In fact most of the concrete
implemented systems that support algebraic specification formalisms are based on
conditional rewriting [29]. For this reason the generalization of the theory of this
thesis to conditional rewriting is very useful in practice. The time bounds of this
research project did not allow a thorough treatment of the generalization to condi-
tional rewriting. We had to confine ourselves to showing that this generalization is
possible, and in fact not difficult.

With conditional rewriting, rules are extended with zero, one or more pairs of
terms. The general shape of a conditional rule being:

\[ \langle l, r, \langle cl_1, cr_1 \rangle, \ldots, \langle cl_n, cr_n \rangle \rangle. \]

The effect of the conditions $\langle cl_1, cr_1 \rangle, \ldots, \langle cl_n, cr_n \rangle$ on the operation of the rule
is that a rewriting operation on a term in which $l$ occurs (into a term in which $l$
is replaced by $r$) is only performed if all conditions hold, after a substitution of
sub-terms for the variables in the condition pairs, sub-terms which emerged in the
matching of $l$. The definition of when exactly a condition does hold is usually defined
in one of the following three ways [46]:
11.3. ADDING CONDITIONS TO RULES

1. A condition holds if both terms of the condition are convertible into each other, based on the entire conditional rewrite system (i.e. both terms are related in the transitive-symmetric closure of the one-step rewrite relation),

2. A condition holds if both terms of the conditions have a common reduct, based on the entire conditional rewrite system,

3. A condition holds if one of the terms of the condition is reducible into the other (which is in normal form), based on the entire conditional rewrite system.

It is easy to see that 3. is a special case of 2. Because of problems with the determination of the convertibility relation, only the variants 2. and 3. are used in practical implementations of conditional rewriting. We will therefore focus on variant 2. for the generalization of our transformation theory. Like with unconditional rewriting we will start with the discussion of ground rewriting.

In section 11.2 (about generalizing ground rewriting to rewriting terms with variables) it was shown that the elementary transformations sometimes have as a consequence that one term with variables in the original domain cannot be covered by one term in the simulating domain. Sometimes two or even more (though a finite number, as was already stated in section 11.2) terms are necessary to cover the generic term set associated with the original term. It is again this phenomenon that causes the complications.

If condition pairs did consist only of pairs of ground terms, the transformation of conditional rules could be done by simply determining the $\phi$-images of both components of each condition (besides eventual rule multiplications due to the form of the left hand side and right hand side patterns of the rules). If conditional rules have conditions with variables and moreover: terms with variables which cannot be represented by one term after a domain transformation, then rules have to be multiplied to support all possible "joining" rewrite sequences between the possible representatives of the original components of the condition. An somewhat abstract example to illustrate this.

**Example 11.2. Multiplication of conditional rules.** Consider the conditional rule:

$$\langle l, r \rangle, \langle cl, cr \rangle,$$

with left hand side and right hand side pattern $l$ and $r$ and condition with components $cl$ and $cr$. If both $cl$ and $cr$ have to be represented by two terms in the simulating domain, say $cl_1, cl_2$ and $cr_1, cr_2$, then the functionality of this rule can in general be covered by four rules. The splitting of $cl$ into $cl_1$ and $cl_2$, and the splitting of $cr$ into $cr_1$ and $cr_2$ has always consequences for some of the variables in $cl$ and $cr$. Variables are the reason for this splitting (section 11.2) and the remedy corresponds essentially to a partition of the $\mu$-value of problem causing variables. It is easy to see now that the functionality of the original rule can be covered by the four rules:

$$\langle l_{11}, r_{11} \rangle, \langle cl_1, cr_1 \rangle,$$
\[(l_{12}, r_{12}), (cl_1, cr_2)\],
\[(l_{21}, r_{21}), (cl_2, cr_1)\],
\[(l_{22}, r_{22}), (cl_2, cr_2)\],
in which \(l_{11}, r_{11}\), etc. are the rule patterns \(l\) and \(r\) with narrower typed variables, corresponding to the types of these variables in \(cl_1, cr_1\), etc.

Obviously the extension of the term rewriting formalism with conditions creates new opportunities for simplifications of the (conditional) rewrite systems. Most of the theory of chapter 9 (transformation strategies) has to be adapted and/or extended to optimally support conditional rewriting. The elaboration of the generalization of the theory to conditional rewriting, the basics of which is now less more than a formalization of what has been said in this section, is left for further research.

The step of generalizing conditional ground rewriting to conditional rewriting of terms with variables is similar to the generalization of unconditional rewriting, which was treated in section 11.2.

### 11.4 More Sophisticated Optimization Strategies

Chapter 9 leaves a lot of directions for further research. The optimization strategies presented in that chapter can be improved in a number of ways. They can be worked out in detail, into efficient concrete algorithms. They can also be extended in order to find more opportunities for the optimization of the various criteria. Theoretical questions like whether or not the optima for the various criteria are unique and whether or not optimal states are decidable are also interesting directions for further research.

### 11.5 The Necessity of the Retrenched Vocabulary Formalism

In this section a thorough justification is given for the introduction of the retrenched vocabulary formalism. As stated earlier (in section 5.4 and section 7.3) there are two main differences between the order sorted signature formalism and the retrenched vocabulary formalism:

- a restriction of the set of constructors which are allowed to act as root constructor of well-formed terms,
- a shortcut between operand and result types of operators, resulting in a restriction function for constructors.
Both aspects are indispensable for the presentation of our theory about syntactic transformations. It is necessary to provide sound arguments for this statement because the introduction of a new basic formalism for the construction of terms needs a thorough justification. The restriction on root constructors is necessary because virtually all elementary transformations have pre-conditions which imply that some constructors are not allowed as root constructor of well-formed terms in some circumstances (all distinguishing and all identifying transformations) or the transformations result in situations in which some constructors are not allowed as root constructor of well-formed terms (the expansion transformation). It is easy to see that these restrictions with regard to root constructors are intrinsic in the transformations. The transformations are only possible if it is possible to have constructors which are allowed to occur inside well-formed terms but not at the root position of well-formed terms.

The shortcut of operand and result types of operators is necessary to keep the size of the transformation descriptions manageable; it is possible to describe the transformations in terms of order sorted signatures (this has in fact been done partly in chapter 7) but these descriptions are far more complicated. Chapter 7 demonstrates the unsuitability of the order sorted signature formalism to describe most of the transformations. The major disadvantage of the conventional order sorted signature formalism for this description is a bad "worst case" behaviour of this formalism with type systems containing many different types and deep hierarchies of types. In certain circumstances, one single distinguishing or contraction transformation can lead to the introduction of many new types (it may double or triple the original number of types) and to a major restructuring of the entire type system in which many or all operators are involved. For the same reason the order sorted signature formalism is little suitable to be a basis for the implementation of the theory (section 11.6).

11.6 Implementing the Theory

The theory that is presented in this thesis can be implemented. Both the processes of overloading resolution (chapter 4) and the derivation of rewrite systems for the elementary syntactic transformations can be implemented easily. This was already suggested by the algorithmic form in which this theory was presented. Implementing the theory is in fact a necessary step in performing further research at the practical applicability of the theory. Transforming large rewrite systems (larger rewrite systems will in general offer more opportunities for simplifications) can only be done mechanically. A few guidelines and suggestions will be given in this section for those who want to implement the ideas of this thesis.

The theory is intended to support (simplify, optimize, ease) rewrite processes, so any implementation of the theory should be part of a system which implements term rewriting. This poses the following problems.

Implementing a new system from scratch is complex because implementing term rewriting processes is already very complex due to aspects like non-determinism and
pattern matching. However, the author believes that an implementation from scratch that allows one to experiment with reasonably large rule sets can be written in a modern high level functional language. It must be a matter of months for an experienced programmer who is familiar with the term rewriting concept to implement basic facilities for input and output of terms and rules, basic facilities for rewriting, and the algorithms that were presented in this thesis.

Extending an existing system that supports term rewriting saves a lot energy but has the disadvantage of a typing system which does not match very well with the elementary transformations; the introduction of the retrenched vocabulary formalism was not without reason. Existing systems that support only many sorted typing are ruled out because many instances of elementary transformations convert many sorted system in systems which are inherently order sorted. This is an easy consequence of the proven fact that well-chosen sequences of elementary transformation can turn an order sorted system into a many sorted system (section 5.4.4 and 9.6.5) and each elementary transformation is fully reversible. Order sorted typing system have the disadvantage that became clear in chapter 7: substantial increases in the number of types and the relations between them.

The author of this thesis designed a functional language which is optimally suited for the implementation of the theory. The language has imperative primitives (assignments, iteration constructs) which enable an easy implementation of the algorithms of chapter 6, and a type system in which types are identified with sets of constructors (and more generally: sets of terms) which supports the typing conception of the retrenched vocabulary formalism. The language is called TERP (TERm Processing) and is described in [62].

11.7 The Research Question

In this section it will be discussed in what way the research question has been answered in the course of the thesis. The research question was:

"The interest is in a general formalism that enables us to simulate typed term rewriting systems with rewrite systems which are in some sense simpler than the original rewrite system. What is an appropriate formalism for the simulation and transformation of typed term rewriting systems?",

simplification hypothetically assumed by:

The suitability of term rewriting systems to formalize subtree replacement techniques is enhanced if it is possible to transform term rewriting systems in systems which are simpler than the original system. A term rewriting system \( TRS_B \) is simpler than a term rewriting system \( TRS_A \) if:

- \( TRS_B \) has less rewrite rules than \( TRS_A \) or:
11.7. THE RESEARCH QUESTION

- the sizes of some (or all) terms in the term domain of TRS_B are less than the sizes of the terms in the term domain of TRS_A or:

- the number of different operators involved in TRS_B is less than the number of different operators involved in TRS_A or:

- TRS_B is a special case of TRS_A with regard to a certain criterion.

The "general formalism" which is the main subject of the research question (chapter 1) is formed by the theory which is presented in chapter 3, 4, 5, 6, 8, 9. Chapter 7 provides a theoretical and chapter 10 a practical justification for the developed theory. The heart of what we will call the "constructive" chapters is chapter 6: the presentation of eight elementary syntactical transformations. Chapter 3 provides a background for the elementary transformations by defining the simulation and transformation concept in general terms, and by formalizing important properties of simulation and transformation processes. Chapter 4 provides a link between the conventional formalism for typed terms (the order sorted signature formalism) and the formalism in terms of which the elementary transformations are defined (the retrenched vocabulary formalism). Chapter 5 is entirely devoted to the presentation of this retrenched vocabulary formalism. A justification for the introduction of this new basic formalism for the construction of terms was subject of section 11.5. In chapter 8 some general ways of composing the elementary transformations are discussed and in chapter 9 particular strategies are presented for the composition of elementary transformations in order to reach a certain simplification goal. Chapter 2 is only a summary of existing theory in the field, chapter 7 and 10 are not constructive in the way like chapters 3, 4, 5, 6, 8, 9 are.

Relating the various parts of the research question to chapters: the "general formalism" consists of the theory of chapter 3, 4, 5, 6, 8, 9; the "simulation" is covered by chapter 3; the "typed terms" are supported by chapter 4, which provides a link to the order sorted signature formalism; the "simplification" is covered by chapter 9, which provides strategies to attain the simplifications that were mentioned in the work hypothesis.

The practical value of the various simplifications is addressed in chapter 10. In this chapter it is shown that there are plenty opportunities for simplifications in practical situations. Usefulness of the simplifications (theoretical or practical usefulness) is assumed in the work-hypothesis. This assumption is very reasonable because of the large number of possible applications of the theory. The achievements of the research that is documented in this dissertation are:

- the development of a number of general simulation techniques and the presentation of some clear examples of the use of these techniques (chapters 3 and 4),

- the development of a new basic formalism for the specification of term sets, in which types are identified with constructor sets (chapter 5),
• the development of eight particular elementary syntactical transformations that allow the restructuring of the shape of terms (chapter 6),

• the development of a generalization of the algebraic homomorphism notion as a basis for the description of the semantic consequences of the eight syntactical transformations (chapter 7),

• an exploration of the transformations space which is generated by the eight elementary transformations (chapter 8),

• the development of some simple strategies for the optimization of certain criteria by the eight elementary transformations (chapter 9).
Bibliography


Summary

In the dissertation a number of techniques for the simulation and transformation of typed term rewriting systems are being presented. The relationship between simulation and transformation is that a transformation will lead to a system that simulates the original system. Three techniques are being introduced: simple step simulation, class step simulation, and composed step simulation. It is formally proven that each of the techniques preserves the for term rewriting systems relevant properties termination and confluence. Besides this, an important class of simple step transformations (transformations that correspond to simple step simulations), is proven to be generated by eight transformations which can be considered as elementary transformations. The eight elementary transformations correspond on semantical level to fundamental abstraction mechanisms. The significance of the distinguished class of simple step transformations is shown. The significance of the other two simulation techniques with regard to the distinguished class is the support of the form in which the eight elementary transformations are defined. This form does not allow multiply declared (overloaded) operators. With class step simulation and composed step simulation systems with multiply declared operators can be simulated by systems without multiply declared operators. The theory, which contains interesting aspects in theoretical as well as in practical respects, is developed from a practical problem formulation. The newly developed theory is fed back to the starting point in the thorough treatment of a practical case.
Samenvatting

In het proefschrift worden een aantal technieken voor de simulatie en transformatie van getypeerde term-herschrijf-systemen gepresenteerd. De relatie tussen simulatie en transformatie is dat transformatie leidt tot een systeem dat het oorspronkelijke systeem simuleert. Er worden drie technieken geïntroduceerd: *simple step simulation*, *class step simulation* en *composed step simulation*. Van iedere techniek wordt het behoud van de voor term-herschrijf-systemen van belang zijnde eigenschappen *termination* en *confluence* formeel bewezen. Daarnaast wordt van een belangrijke klasse van *simple step transformations* (transformaties die corresponderen met *simple step simulations*) bewezen dat ze gegenereerd wordt door acht als elementair te beschouwen transformaties. De acht elementaire syntactische transformaties corresponderen op semantisch niveau met fundamentele abstractiemechanismen. Het belang van de onderscheiden klasse van *simple step transformations* wordt aangetoond. Het belang van de andere twee simulatie technieken in verband met de onderscheiden klasse is de ondersteuning van de vorm waarin de acht elementaire transformaties gedefinieerd worden. Deze vorm staat geen meervoudig gedeclareerde (overloaded) operatoren toe. Met behulp van *class step simulation* en *composed step simulation* kunnen systemen met meervoudig gedeclareerde operatoren gesimuleerd worden door systemen zonder meervoudig gedeclareerde operatoren. De theorie, die zowel in theoretisch als in praktisch opzicht interessante aspecten heeft, is ontwikkeld vanuit een praktische probleemstelling. De nieuwe theorie wordt teruggekoppeld naar het uitgangspunt in de uitgebreide behandeling van een case.
Curriculum Vitae

Naam: Jan Verheul.

Geboren: 24 april 1964.


Februari 1991: Doctoraal examen Informatica (cum laude), Universiteit Twente, Enschede.


Februari 1989 - december 1990: Technisch medewerker aan de Universiteit Twente (50 % deeltijd).