Term *Re*writing and *All That*

Franz Baader
Tobias Nipkow

Term *Re*writing and *All That*

This is the first English language textbook offering a unified and self-contained introduction to the field of term rewriting. It covers all the basic material (abstract reduction systems, termination, confluence, completion, and combination problems), but also some important and closely connected subjects: universal algebra, unification theory, and Gröbner bases. The main algorithms are presented both informally and as programs in the functional language Standard ML (an appendix contains a quick and easy introduction to ML). Certain crucial algorithms like unification and congruence closure are covered in more depth and efficient Pascal programs are developed. The book contains many examples and over 170 exercises.

This text is also an ideal reference book for professional researchers: results that have been spread over many conference and journal articles are collected together in a unified notation, detailed proofs of almost all theorems are provided, and each chapter closes with a guide to the literature.

Term *Re*writing and *All That*

Franz Baader
RWTH, Aachen

and

Tobias Nipkow
Technische Universität, München

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521455206

First published 1998
First paperback edition 1999

Typeset by the authors

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication data

Baader, Franz.
Term rewriting and all that / Franz Baader and Tobias Nipkow.
p. cm.
Includes bibliographical references and index.
ISBN 0 521 45520 0 (hc : alk. paper)
1. Rewriting systems (Computer science). I. Nipkow, Tobias, 1958–
II. Title.
QA267.B314 1998
005.13′1–dc21 97-28286 CIP

ISBN-13 978-0-521-45520-6 hardback
ISBN-10 0-521-45520-0 hardback

ISBN-13 978-0-521-77920-3 paperback
ISBN-10 0-521-77920-0 paperback

Transferred to digital printing 2006

Contents

Preface *page* ix

1 Motivating Examples 1

2 Abstract Reduction Systems 7

2.1 Equivalence and reduction 7

2.2 Well-founded induction 13

2.3 Proving termination 16

2.4 Lexicographic orders 18

2.5 Multiset orders 21

2.6 Orders in ML 26

2.7 Proving confluence 28

2.8 Bibliographic notes 33

3 Universal Algebra 34

3.1 Terms, substitutions, and identities 34

3.2 Algebras, homomorphisms, and congruences 44

3.3 Free algebras 47

3.4 Term algebras 49

3.5 Equational classes 49

4 Equational Problems 58

4.1 Deciding $\approx_E$ 59

4.2 Term rewriting systems 61

4.3 Congruence closure 62

4.4 Congruence closure on graphs 65

4.5 Syntactic unification 71

4.6 Unification by transformation 73

4.7 Unification and term rewriting in ML 79

4.8 Unification of term graphs 82

4.9 Bibliographic notes 91

5 Termination 93
5.1 The decision problem 93
5.2 Reduction orders 101
5.3 The interpretation method 104
5.4 Simplification orders 111
5.5 Bibliographic notes 131

6 Confluence 134
6.1 The decision problem 134
6.2 Critical pairs 135
6.3 Orthogonality 145
6.4 Beyond orthogonality 151
6.5 Bibliographic notes 157

7 Completion 158
7.1 The basic completion procedure 160
7.2 An improved completion procedure 164
7.3 Proof orders 172
7.4 Huet's completion procedure 178
7.5 Huet's completion procedure in ML 182
7.6 Bibliographic notes 184

8 Gröbner Bases and Buchberger's Algorithm 187
8.1 The ideal membership problem 187
8.2 Polynomial reduction 189
8.3 Gröbner bases 193
8.4 Buchberger's algorithm 196
8.5 Bibliographic notes 198

9 Combination Problems 200
9.1 Basic notions 200
9.2 Termination 202
9.3 Confluence 207
9.4 Combining word problems 211
9.5 Bibliographic notes 222

10 Equational Unification 223
10.1 Basic definitions and results 224
10.2 Commutative functions 230
10.3 Associative and commutative functions 236
10.4 Boolean rings 250
10.5 Bibliographic notes 262

11 Extensions 265
11.1 Rewriting modulo equational theories 265
11.2 Ordered rewriting 267
11.3 Conditional identities and conditional rewriting 269
11.4 Higher-order rewrite systems 270
11.5 Reduction strategies 271
11.6 Narrowing 273
Appendix 1 **Ordered Sets** 276
Appendix 2 **A Bluffer's Guide to** ML 278
Bibliography 284
Index 297

Preface

Term rewriting is a branch of theoretical computer science which combines elements of logic, universal algebra, automated theorem proving and functional programming. Its foundation is equational logic. What distinguishes term rewriting from equational logic is that equations are used as *directed* replacement rules, i.e. the left-hand side can be replaced by the right-hand side, but not vice versa. This constitutes a Turing-complete computational model which is very close to functional programming. It has applications in algebra (e.g. Boolean algebra, group theory and ring theory), recursion theory (what is and is not computable with certain sets of rewrite rules), software engineering (reasoning about equationally defined data types such as numbers, lists, sets etc.), and programming languages (especially functional and logic programming). In general, term rewriting applies in any context where efficient methods for reasoning with equations are required.

To date, most of the term rewriting literature has been published in specialist conference proceedings (especially *Rewriting Techniques and Applications* and *Automated Deduction* in Springer's LNCS series) and journals (e.g. *Journal of Symbolic Computation* and *Journal of Automated Reasoning*). In addition, several overview articles provide introductions into the field, and references to the relevant literature [141, 74, 204]. This is the first English book devoted to the theory and applications of term rewriting. It is ambitious in that it tries to serve two masters:

- The researcher, who needs a unified theory that covers, in detail and in a single volume, material that has previously only been collected in overview articles, and whose technical details are spread over the literature.
- The teacher or student, who needs a readable textbook in an area where there is hardly any literature for the non-specialist.

Our choice of material is fairly canonical: abstract reduction systems and universal algebra (the foundation), word problems (the motivation), unification (a central algorithm), termination, confluence and completion (the *sine qua non* of term rewriting). The inclusion of combination problems is also uncontroversial, except maybe for the rather technical topic of combining word problems. Two further topics show our own preferences and are not strictly core material: equational unification is included because of its significance for rewriting based theorem provers, Gröbner bases because they form an essential link between term rewriting and computer algebra.

Prerequisites are minimal: readers who have taken introductory courses such as discrete mathematics, (linear) algebra, or theoretical computer science are well equipped for this book. The basic notions of ordered sets are summarized in an appendix.

How to teach this book

The diagram below shows the dependencies between the different sections of the book.

$$\begin{tikzcd}
 & & 2 \arrow[d] & & & \\
 & 3.2\text{–}3.5 & 3.1 \arrow[l] \arrow[d] & & & \\
4.3\text{–}4.4 & & 4.1\text{–}4.2,\ 4.5\text{–}4.6 \arrow[ll] \arrow[r] \arrow[drr] \arrow[d] & 4.8 & & \\
 & & 5 \arrow[d] & & 10 \arrow[r] & 11.1 \\
 & 6.4 & 6.1\text{–}6.3 \arrow[l] \arrow[r] \arrow[drr] \arrow[d] & 11.2\text{–}11.6 & & \\
 & & 7.1 \arrow[dl] \arrow[dr] & & 9 & \\
 & 7.2\text{–}7.4 & & 8 & &
\end{tikzcd}$$

An introductory undergraduate course should cover the trunk of the above tree. To give the students a more algorithmic understanding of completion, it is helpful also to introduce Huet's completion procedure (7.4) without formally justifying its correctness. The course should conclude with 11.2–11.6. A more advanced introduction at graduate level would also include 4.3–4.4, 4.8, 6.4, 7.2–7.4, 9.1–9.3, and (initial segments of) 10. For a mathematically oriented audience, 3.2–3.5 is mandatory and 8 contains an excellent application of rewriting methods in mathematics.

Chapter 2 on abstract reduction systems is the foundation that term rewriting rests on. Nevertheless we recommend not to teach this chapter *en bloc* but to interleave it with the rest of the book. Only Section 2.1 needs to be covered right at the start. The dependency of the remaining sections is as follows:

$$2.2\text{–}2.5 \longrightarrow 5 \longrightarrow 2.7 \longrightarrow 6.$$

This groups together the abstract and concrete treatments of termination (2.2–2.5 and 5) and confluence (2.7 and 6).

Chapter 5 on termination has a special status in the dependency diagram. It is not the case that all of Chapter 5 is a prerequisite for the remainder of the book. In fact, almost the opposite is the case: one could read most of the remainder quite happily, except that one would not be able to follow particular termination arguments. However, due to the overall importance of termination, we recommend that students should be exposed at least to 5.1–5.3 and possibly one of the simplification orders in 5.4. The general theory of simplification orders should be reserved for a graduate-level course.

A final word of warning. A book also aimed at researchers is written with a higher level of formality than a pure textbook. In places, the formal rigour of the book needs to be adjusted to the requirements of the classroom.

The rôle of ML

Most of the theory in this book is constructive. Either we explicitly deal with particular algorithms, e.g. unification, or the proof of some theorem is essentially an algorithm, e.g. a decision procedure. We find that many computer science students take more easily to logical formalisms once they understand how to represent formulae as data structures and how to transform them. Therefore we have tried to accompany every major algorithm in this book by an implementation. As an implementation language we have chosen ML: functional languages are closest to our algorithms and ML is one of its best-known representatives. For those readers not familiar with ML, a concise summary of the core of the language is provided as an appendix.

It should be emphasized that our ML programs are strictly added value: they reside in separate sections and are not required for an understanding of the main text (although we believe that their study enhances this understanding).

We should also point out that the programs are intentionally unoptimized.

They are written for clarity rather than efficiency. Nevertheless they cope well with small to medium sized examples. Their simplicity makes them an ideal vehicle for further developments, and we encourage our readers to experiment with them. They are available on the internet at

```
http://www4.informatik.tu-muenchen.de/~nipkow/
```

Acknowledgments

David Basin, Eric Domenjoud, Harald Ganzinger, Bernhard Gramlich, Henrik Linnestad, Aart Middeldorp, Monica Nesi, Vincent van Oostrom, Larry Paulson, Manfred Schmidt-Schauß, Klaus Schulz, Wayne Snyder, Cesare Tinelli, and Markus Wenzel read individual chapters and commented extensively on them. In particular Aart Middeldorp's amazing scrutiny uncovered a number of embarrassing mistakes.

Michael Hanus, Maribel Fernández and Femke van Raamsdonk provided additional comments.

Can A. Albayrak and Volker Braun produced first versions of some of the figures.

The DFG funded a sabbatical of the second author at Cambridge University Computer Laboratory where Larry Paulson greatly contributed to a very productive four months.

Alison Woollatt of CUP provided essential TEXpertise. David Tranah, our very patient editor, suggested the title.

We wish to thank them all.

1
Motivating Examples

Equational reasoning is concerned with a rather restricted class of first-order languages: the only predicate symbol is equality. It is, however, at the heart of many problems in mathematics and computer science, which explains why developing specialized methods and tools for this type of reasoning is very popular and important. For example, in mathematics one often defines classes of algebras (such as groups, rings, etc.) by giving defining identities (which state associativity of the group operation, etc.). In this context, it is important to know which other identities can be derived from the defining ones. In algebraic specification, new operations are defined from given ones by stating characteristic identities that must hold for the defined operations. As a special case we have functional programs where functions are defined by recursion equations.

For example, assume that we want to define addition of natural numbers using the constant 0 and the successor function s. This can be done with the identities†

$$\begin{aligned} x+0 &\approx x, \\ x+s(y) &\approx s(x+y). \end{aligned}$$

By applying these identities, we can calculate the sum of 1 (encoded as $s(0)$) and 2 (encoded as $s(s(0))$):

$$s(0)+s(s(0)) \approx s(s(0)+s(0)) \approx s(s(s(0))+0) \approx s(s(s(0))).$$

In this calculation, we have interpreted the identities as rewrite rules that tell us how a subterm of a given term can be replaced by another term.

This brings us to one of the key notions of this book, namely **term rewriting systems**. What do we mean by **terms**? They are built from **variables**,

† Throughout this book, we use $\approx$ for identities to make a clear distinction between the object level sign for identity and our use of $=$ for equality on the meta-level.

constant symbols, and **function symbols**. In the above example, $+$ is a binary function symbol, s is a unary function symbol, 0 is a constant symbol, and x, y are variables. Examples of terms over these symbols are 0, x, $s(s(0))$, $x + s(0)$, $s(s(s(0)) + 0)$. In our example calculation, we have used the identities only from left to right, but in general, identities can be applied in both directions.

In the following, we give two examples that illustrate some of the key issues arising in connection with identities and rewrite systems, and which will be treated in detail in this book. In the first example, the rewrite rules are intended to be used only in one direction (which is expressed by writing $\rightarrow$ instead of $\approx$). This is an instance of rewriting as a computation mechanism. In the second, we consider the identities defining groups, which are intended to be used in both directions. This is an instance of rewriting as a deduction mechanism.

Symbolic Differentiation

We consider symbolic differentiation of arithmetic expressions that are built with the operations $+$, $*$, the indeterminates X, Y, and the numbers $0, 1$. For example, $((X+X)*Y)+1$ is an admissible expression. These expressions can be viewed as terms that are built from the constant symbols 0, 1, X, and Y, and the binary function symbols $+$ and $*$. For the partial derivative with respect to X, we introduce the additional (unary) function symbol D_X. The following rules are (some of the) well-known rules for computing the derivative:

$$\begin{array}{llcl}
(\mathrm{R1}) & D_X(X) & \rightarrow & 1, \\
(\mathrm{R2}) & D_X(Y) & \rightarrow & 0, \\
(\mathrm{R3}) & D_X(u+v) & \rightarrow & D_X(u) + D_X(v), \\
(\mathrm{R4}) & D_X(u*v) & \rightarrow & (u * D_X(v)) + (D_X(u) * v).
\end{array}$$

In terms like $D_X(u+v)$, the symbols u and v are variables, with the intended meaning that they can be replaced by arbitrary expressions.† Thus, rule (R3) can be applied to terms having the same pattern as the left-hand side, i.e. a D_X followed by a $+$-expression.

Starting with the term $D_X(X*X)$, the rules (R1)–(R4) lead to the possible reductions depicted in Fig. 1.1. We can use this example to illustrate two of the most important properties of term rewriting systems:

† These variables should not be confused with the indeterminates X, Y of the arithmetic expressions, which are constant symbols.

$$\begin{tikzcd}
& D_X(X * X) \arrow[d, "\mathrm{R4}"] & \\
& (X * D_X(X)) + (D_X(X) * X) \arrow[dl, "\mathrm{R1}"] \arrow[dr, "\mathrm{R1}"] & \\
(X * 1) + (D_X(X) * X) \arrow[dr, "\mathrm{R1}"] & & (X * D_X(X)) + (1 * X) \arrow[dl, "\mathrm{R1}"'] \\
& (X * 1) + (1 * X) &
\end{tikzcd}$$

Fig. 1.1. Symbolic differentiation of the expression $D_X(X * X)$.

Termination: Is it always the case that after finitely many rule applications we reach an expression to which no more rules apply? Such an expression is then called a **normal form**.

For the rules (R1)–(R4) this is the case. It is, however, not completely trivial to show this because rule (R4) leads to a considerable increase in the size of the expression.

An example of a non-terminating rule is

$$u + v \rightarrow v + u,$$

which expresses commutativity of addition. The sequence $(X * 1) + (1 * X) \rightarrow (1 * X) + (X * 1) \rightarrow (X * 1) + (1 * X) \rightarrow \ldots$ is an example for an infinite chain of applications of this rule. Of course, non-termination need not always be caused by a single rule; it could also result from the interaction of several rules.

Confluence: If there are different ways of applying rules to a given term t, leading to different derived terms t_1 and t_2, can t_1 and t_2 be joined, i.e. can we always find a common term s that can be reached both from t_1 and from t_2 by rule application?

In Fig. 1.1 this is the case, and more generally, one can prove (but how?) that (R1)–(R4) are confluent. This shows that the symbolic differentiation of a given expression always leads to the same derivative (i.e. the term to which no more rules apply), independent of the strategy for applying rules.

If we add the simplification rule

$$\text{(R5)} \quad u + 0 \to u$$

to (R1)–(R4), we lose the confluence property (see Fig. 1.2).

$$\begin{tikzcd}
& D_X(X+0) \arrow[dl, "\text{R5}"] \arrow[dr, "\text{R3}"] & \\
D_X(X) \arrow[d, "\text{R1}"] & & D_X(X) + D_X(0) \arrow[d, "\text{R1}"] \\
1 & & 1 + D_X(0)
\end{tikzcd}$$

Fig. 1.2. $D_X(X)$ and $D_X(X) + D_X(0)$ cannot be joined.

In our example, non-confluence of (R1)–(R5) can be overcome by adding the rule $D_X(0) \to 0$. More generally, one can ask whether this is always possible, i.e. can we always make a non-confluent system confluent by adding implied rules (**completion** of term rewriting systems).

Because of their special form, the rules (R1)–(R4) constitute a functional program (on the left-hand side, the defined function D_X occurs only at the very outside). Termination of the rules means that D_X is a total function. Confluence of the rules means that the result of a computation is independent of the evaluation strategy. Confluence of (R1)–(R4) is not a lucky coincidence. We will prove that all term rewriting systems that constitute functional programs are confluent.

Group Theory

Let $\circ$ be a binary function symbol, i be a unary function symbol, e be a constant symbol, and x, y, z be variable symbols. The class of all groups is defined by the identities

$$\begin{array}{lrcl}
\text{(G1)} & (x \circ y) \circ z & \approx & x \circ (y \circ z), \\
\text{(G2)} & e \circ x & \approx & x, \\
\text{(G3)} & i(x) \circ x & \approx & e,
\end{array}$$

i.e. a set G equipped with a binary operation $\circ$, a unary operation i, and containing an element e is a group iff the operations satisfy the identities (G1)–(G3). Identity (G3) states only that for every group element g, the element $i(g)$ is a left-inverse of g with respect to the left-unit e. The identities (G1)–(G3) can be used to show that this left-inverse is also a right-inverse. In fact, using these identities, the term e can be transformed into the term $x \circ i(x)$:

$$\begin{aligned}
e &\overset{\text{G3}}{\approx} i(x \circ i(x)) \circ (x \circ i(x)) \\
&\overset{\text{G2}}{\approx} i(x \circ i(x)) \circ (x \circ (e \circ i(x))) \\
&\overset{\text{G3}}{\approx} i(x \circ i(x)) \circ (x \circ ((i(x) \circ x) \circ i(x))) \\
&\overset{\text{G1}}{\approx} i(x \circ i(x)) \circ ((x \circ (i(x) \circ x)) \circ i(x)) \\
&\overset{\text{G1}}{\approx} i(x \circ i(x)) \circ (((x \circ i(x)) \circ x) \circ i(x)) \\
&\overset{\text{G1}}{\approx} i(x \circ i(x)) \circ ((x \circ i(x)) \circ (x \circ i(x))) \\
&\overset{\text{G1}}{\approx} (i(x \circ i(x)) \circ (x \circ i(x))) \circ (x \circ i(x)) \\
&\overset{\text{G3}}{\approx} e \circ (x \circ i(x)) \\
&\overset{\text{G2}}{\approx} x \circ i(x).
\end{aligned}$$

This example illustrates that it is nontrivial to find such derivations, i.e. to solve the so-called **word problem** for sets of identities: given a set of identities E and two terms s and t, is it possible to transform the term s into the term t, using the identities in E as rewrite rules that can be applied in both directions?

One possible way of approaching this problem is to consider the identities as uni-directional rewrite rules:

$$\begin{array}{lrcl}
\text{(RG1)} & (x \circ y) \circ z & \rightarrow & x \circ (y \circ z), \\
\text{(RG2)} & e \circ x & \rightarrow & x, \\
\text{(RG3)} & i(x) \circ x & \rightarrow & e.
\end{array}$$

The basic idea is that the identities are only applied in the direction that "simplifies" a given term. One is now looking for normal forms, i.e. terms to which no more rules apply. In order to decide whether the terms s and t are equivalent (i.e. can be transformed into each other by applying identities in both directions), we use the uni-directional rewrite rules to reduce s to a normal form $\widehat{s}$ and t to a normal form $\widehat{t}$. Then we check whether $\widehat{s}$ and $\widehat{t}$ are syntactically equal. There are, however, two problems that must be overcome before this method for deciding the word problem can be applied:

- Equivalent terms can have distinct normal forms. In our example, both $x \circ i(x)$ and e are normal forms with respect to (RG1)–(RG3), and we have shown that they are equivalent. However, the above method for deciding the word problem would fail because it would find that the normal forms of $x \circ i(x)$ and e are distinct.

- Normal forms need not exist: the process of reducing a term may lead to an infinite chain of rule applications.

We will see that termination and confluence are the important properties that ensure existence and uniqueness of normal forms. If a given set of identities leads to a non-confluent rewrite system, we do not have to give up. We can again apply the idea of completion to extend the rewrite system to a confluent one. In the case of groups, a confluent and terminating extension of (RG1)–(RG3) exists (see Exercise 7.12 on page 184).

2
Abstract Reduction Systems

This chapter is concerned with the abstract treatment of reduction, where reduction is synonymous with the traversal of some directed graph, the stepwise execution of some computation, the gradual transformation of some object (e.g. a term), or any similar step by step activity. Mathematically this means we are simply talking about binary relations. An **abstract reduction system** is a pair $(A, \rightarrow)$, where the **reduction** $\rightarrow$ is a binary relation on the set A, i.e. $\rightarrow \subseteq A \times A$. Instead of $(a, b) \in \rightarrow$ we write $a \rightarrow b$.

The term "reduction" has been chosen because in many applications something decreases with each reduction step, but cannot decrease forever. Yet this need not be the case, as witnessed by the reduction $0 \rightarrow 1 \rightarrow 2 \rightarrow \cdots$.

Unless noted otherwise, all our discussions take place in the context of some arbitrary but fixed abstract reduction system $(A, \rightarrow)$.

2.1 Equivalence and reduction

We can view reduction in two ways: the first is as a directed computation, which, starting from some point a_0, tries to reach a normal form by following the reduction $a_0 \rightarrow a_1 \rightarrow \cdots$. This corresponds to the idea of program evaluation. Or we may consider $\rightarrow$ merely as a description of $\overset{*}{\leftrightarrow}$, where $a \overset{*}{\leftrightarrow} b$ means that there is a path between a and b where the arrows can be traversed in both directions, for example, as in $a_0 \leftarrow a_1 \rightarrow a_2 \leftarrow a_3$. This corresponds to the idea of identities which can be used in both directions. The key question here is to decide if two elements a and b are **equivalent**, i.e. if $a \overset{*}{\leftrightarrow} b$ holds. Settling this question by an undirected search along both $\rightarrow$ and $\leftarrow$ is bound to be expensive. Wouldn't it be nice if we could decide equivalence by reducing both a and b to their normal forms and testing if the normal forms are identical? As explained in the first chapter, this idea is only going to work if reduction terminates and normal forms are unique.

Formally, we talk about *termination* and *confluence* of reduction, and the study of these two notions is one of the central themes of this book.

2.1.1 Basic definitions

In the sequel, we define a great many symbols, not all of which will be put to immediate use. Therefore you may treat these definitions as a table of relevant notions which can be consulted when necessary.

Given two relations $R \subseteq A \times B$ and $S \subseteq B \times C$, their **composition** is defined by

$$R \circ S := \{(x, z) \in A \times C \mid \exists y \in B.\ (x, y) \in R \land (y, z) \in S\}$$

Definition 2.1.1 We are particularly interested in composing a reduction with itself and define the following notions:

$$\begin{array}{lcll}
\xrightarrow{0} & := & \{(x,x) \mid x \in A\} & \textbf{identity} \\
\xrightarrow{i+1} & := & \xrightarrow{i} \circ \rightarrow & (i+1)\textbf{-fold composition}, i \geq 0 \\
\xrightarrow{+} & := & \bigcup_{i>0} \xrightarrow{i} & \textbf{transitive closure} \\
\xrightarrow{*} & := & \xrightarrow{+} \cup \xrightarrow{0} & \textbf{reflexive transitive closure} \\
\xrightarrow{=} & := & \rightarrow \cup \xrightarrow{0} & \textbf{reflexive closure} \\
\xrightarrow{-1} & := & \{(y,x) \mid x \rightarrow y\} & \textbf{inverse} \\
\leftarrow & := & \xrightarrow{-1} & \textbf{inverse} \\
\leftrightarrow & := & \rightarrow \cup \leftarrow & \textbf{symmetric closure} \\
\xleftrightarrow{+} & := & (\leftrightarrow)^{+} & \textbf{transitive symmetric closure} \\
\xleftrightarrow{*} & := & (\leftrightarrow)^{*} & \textbf{reflexive transitive symmetric closure}
\end{array}$$

Some remarks are in order:

1. Notations like $\xrightarrow{*}$ and $\leftarrow$ only work for arrow-like symbols. In the case of arbitrary relations $R \subseteq A \times A$ we write R^*, R^{-1} etc.
2. Some of the constructions can also be expressed nicely in terms of *paths*:
   $x \xrightarrow{n} y$ if there is a path of length n from x to y,
   $x \xrightarrow{*} y$ if there is some finite path from x to y,
   $x \xrightarrow{+} y$ if there is some finite nonempty path from x to y.
3. The word **closure** has a precise meaning: the P closure of R is the least set with property P which contains R. For example, $\xrightarrow{*}$, the reflexive transitive closure of $\rightarrow$, is the least reflexive and transitive relation which contains $\rightarrow$. Note that for arbitrary P and R, the P closure of R need not exist, but in the above cases they always do because reflexivity, transitivity and symmetry are closed under arbitrary intersections. In

such cases the P closure of R can be defined directly as the intersection of all sets with property P which contain R.

4. It is easy to show that $\overset{*}{\leftrightarrow}$ is the least equivalence relation containing $\rightarrow$.

Let us add some terminology to this notation:

1. x is **reducible** iff there is a y such that $x \rightarrow y$.
2. x is **in normal form** (**irreducible**) iff it is not reducible.
3. y is **a normal form of** x iff $x \overset{*}{\rightarrow} y$ and y is in normal form. If x has a uniquely determined normal form, the latter is denoted by $x{\downarrow}$.
4. y is a **direct successor** of x iff $x \rightarrow y$.
5. y is a **successor** of x iff $x \overset{+}{\rightarrow} y$
6. x and y are **joinable** iff there is a z such that $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$, in which case we write $x \downarrow y$.

Example 2.1.2

1. Let $A := \mathbb{N} - \{0, 1\}$ and $\rightarrow := \{(m, n) \mid m > n \text{ and } n \text{ divides } m\}$. Then
   (a) m is in normal form iff m is prime.
   (b) p is a normal form of m iff p is a prime factor of m.
   (c) $m \downarrow n$ iff m and n are not relatively prime.
   (d) $\overset{+}{\rightarrow} = \rightarrow$ because $>$ and "divides" are already transitive.
   (e) $\overset{*}{\leftrightarrow} = A \times A$.
2. Let $A := \{a, b\}^*$ (the set of words over the alphabet $\{a, b\}$) and $\rightarrow := \{(ubav, uabv) \mid u, v \in A\}$. Then
   (a) w is in normal form iff w is sorted, i.e. of the form a^*b^*.
   (b) Every w has a unique normal form $w{\downarrow}$, the result of sorting w.
   (c) $w_1 \downarrow w_2$ iff $w_1 \overset{*}{\leftrightarrow} w_2$ iff w_1 and w_2 contain the same number of as and bs.

Finally we come to some of the central notions of this book.

Definition 2.1.3 A reduction $\rightarrow$ is called

Church-Rosser†	iff	$x \overset{*}{\leftrightarrow} y \;\Rightarrow\; x \downarrow y$ (see Fig. 2.1).
confluent	iff	$y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \;\Rightarrow\; y_1 \downarrow y_2$ (see Fig. 2.1).
terminating	iff	there is no infinite descending chain $a_0 \rightarrow a_1 \rightarrow \cdots$
normalizing	iff	every element has a normal form.
convergent	iff	it is both confluent and terminating.

Both reductions in Example 2.1.2 terminate, but only the second one is Church-Rosser and confluent.

† Alonzo Church and J. Barkley Rosser proved that the λ-calculus has this property [51].

$$\begin{tikzcd} x \arrow[rr, leftrightarrow, "*"] \arrow[dr, dashed, "*"'] & & y \arrow[dl, dashed, "*"] \\ & z & \end{tikzcd} \qquad \begin{tikzcd} x \arrow[r, "*"] \arrow[d, "*"'] & y_1 \arrow[d, dashed, "*"] \\ y_2 \arrow[r, dashed, "*"'] & z \end{tikzcd} \qquad \begin{tikzcd} x \arrow[r] \arrow[d, "*"'] & y_1 \arrow[d, dashed, "*"] \\ y_2 \arrow[r, dashed, "*"'] & z \end{tikzcd}$$

Fig. 2.1. Church-Rosser property, confluence and semi-confluence.

Remarks:

1. The diagrams in Fig. 2.1 have a precise meaning and are used throughout the book in this manner: solid arrows represent universal and dashed arrows existential quantification; the whole diagram is an implication of the form $\forall \overline{x}.\ P(\overline{x}) \Rightarrow \exists \overline{y}.\ Q(\overline{x}, \overline{y})$. For example, the confluence diagram becomes $\forall x, y_1, y_2.\ y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \Rightarrow \exists z.\ y_1 \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y_2$.
2. Because $x \downarrow y$ implies $x \overset{*}{\leftrightarrow} y$, the Church-Rosser property can also be phrased as an equivalence: $x \overset{*}{\leftrightarrow} y \Leftrightarrow x \downarrow y$.
3. Any terminating relation is normalizing, but the converse is not true, as the example in Fig 2.2 shows.

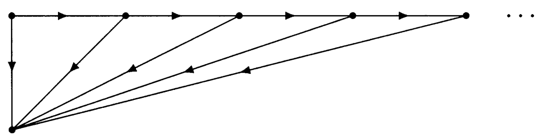

Fig. 2.2. Confluent, normalizing and acyclic but not terminating.

Thus we have come back to our initial motivation: the Church-Rosser property is exactly what we were looking for, namely the ability to test equivalence by the search for a common successor. We will now see how it relates to termination and confluence.

2.1.2 Basic results

It turns out that the Church-Rosser property and confluence coincide. The fact that any Church-Rosser relation is confluent is almost immediate, and the reverse implication has a beautiful diagrammatic proof which is shown in Fig. 2.3. It is based on the observation that any equivalence $x \overset{*}{\leftrightarrow} y$ can be

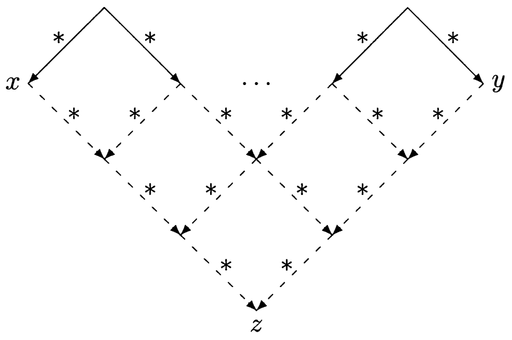

Fig. 2.3. Confluence implies the Church-Rosser property.

written as a series of peaks as in the top of the diagram. Now you can use confluence to complete the diagram from the top to the bottom. The formal proof below yields some additional information by involving an intermediate property:

Definition 2.1.4 A relation $\rightarrow$ is **semi-confluent** (Fig. 2.1) iff

$$y_1 \leftarrow x \overset{*}{\rightarrow} y_2 \;\Rightarrow\; y_1 \downarrow y_2.$$

Although semi-confluence looks weaker than confluence, it turns out to be equivalent:

Theorem 2.1.5 *The following conditions are equivalent:*

1. $\rightarrow$ has the Church-Rosser property.
2. $\rightarrow$ is confluent.
3. $\rightarrow$ is semi-confluent.

Proof We show that the implications $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$ hold.

($1 \Rightarrow 2$) If $\rightarrow$ has the Church-Rosser property and $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2$ then $y_1 \overset{*}{\leftrightarrow} y_2$ and hence, by the Church-Rosser property, $y_1 \downarrow y_2$, i.e. $\rightarrow$ is confluent.

($2 \Rightarrow 3$) Obviously any confluent relation is semi-confluent.

($3 \Rightarrow 1$) If $\rightarrow$ is semi-confluent and $x \overset{*}{\leftrightarrow} y$ then we show $x \downarrow y$, i.e. the Church-Rosser property, by induction on the length of the chain $x \overset{*}{\leftrightarrow} y$. If $x = y$, this is trivial. If $x \overset{*}{\leftrightarrow} y \leftrightarrow y'$ we know $x \downarrow y$ by induction hypothesis, i.e. $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$ for some suitable z. We show $x \downarrow y'$ by case distinction:

$y \leftarrow y'$: $x \downarrow y'$ follows directly from $x \downarrow y$.
$y \rightarrow y'$: semi-confluence implies $z \downarrow y'$ and hence $x \downarrow y'$.

The reasoning is displayed graphically in Fig. 2.4. □

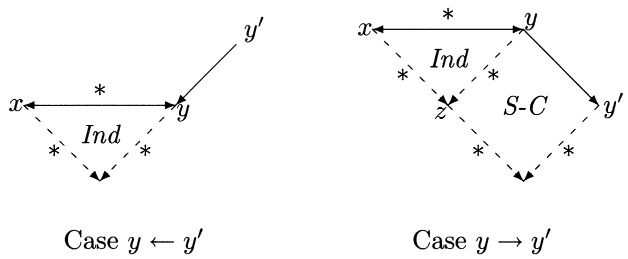

Fig. 2.4. Semi-confluence implies the Church-Rosser property.

This theorem has some easy consequences:

Corollary 2.1.6 *If $\rightarrow$ is confluent and $x \overset{*}{\leftrightarrow} y$ then*

1. *$x \overset{*}{\rightarrow} y$ if y is in normal form, and*
2. *$x = y$ if both x and y are in normal form.*

Now we know that for confluent relations, two elements are equivalent iff they are joinable. Of course the test for joinability can be difficult (and even undecidable) if the relation does not terminate: given two elements which are not joinable, when should you stop the search for a common successor in case of an infinite reduction starting from one of the two elements, as in the following example?

$$\begin{array}{ccccccc} a_0 & \rightarrow & a_1 & \rightarrow & a_2 & \rightarrow & \cdots, \\ b_0 & \rightarrow & b_1 & \rightarrow & b_2 & \rightarrow & \cdots. \end{array}$$

It turns out that normalization suffices for determining joinability. To see this, let us explore the relationship between termination, normalization, confluence and the uniqueness of normal forms.

Fact 2.1.7 *If $\rightarrow$ is confluent, every element has at most one normal form.*

Since every element has at least one normal form if $\rightarrow$ is normalizing, it follows that for confluent and normalizing relations every element x has exactly one normal form which we write $x{\downarrow}$:

Lemma 2.1.8 *If $\rightarrow$ is normalizing and confluent, every element has a unique normal form.*

Having established under what conditions the notation $x{\downarrow}$ is well-defined, we immediately obtain our main theorem:

Theorem 2.1.9 *If $\rightarrow$ is confluent and normalizing then $x \overset{*}{\leftrightarrow} y \;\Leftrightarrow\; x{\downarrow} = y{\downarrow}$.*

Proof The $\Leftarrow$-direction is trivial. Conversely, if $x \overset{*}{\leftrightarrow} y$ then $x{\downarrow} \overset{*}{\leftrightarrow} y{\downarrow}$ and hence $x{\downarrow} = y{\downarrow}$ by Corollary 2.1.6. $\square$

Thus we have finally arrived at a very goal-directed equivalence test: simply check if the normal forms of both elements are identical. Provided normal forms are computable and identity is decidable, equivalence also becomes decidable.

Many authors prefer to work with termination instead of normalization and state Theorem 2.1.9 with "convergent" instead of "confluent and normalizing". Although normalization suffices for finding normal forms, it means that breadth-first rather than depth-first search may be required, for example in Fig. 2.2. For this reason we will also concentrate on termination rather than normalization in the sequel.

Exercises

2.1 Which closure operations commute? Find a proof or counterexample:

(a) Is the reflexive closure of the transitive closure the same as the transitive closure of the reflexive closure, i.e. are $(\overset{+}{\to})^=$ and $(\overset{=}{\to})^+$ the same and do they coincide with $\overset{*}{\to}$?

(b) What about the transitive and the symmetric closure? Do $(\leftrightarrow)^+$ and $(\overset{+}{\to}) \cup (\overset{+}{\to})^{-1}$ coincide?

2.2 Show that $\to$ is confluent and normalizing iff every element has a unique normal form.

2.3 Find a reduction $\to$ on $\mathbb{N}$ such that $\to$ is decidable but it is undecidable if some n is in normal form.

2.2 Well-founded induction

This section introduces the important proof principle of well-founded induction and shows that it is enjoyed by all terminating relations. As a motivation, recall the principle of induction for natural numbers: a property $P(n)$ holds for all natural numbers n if we can show that $P(n)$ holds under the (induction) hypothesis that $P(m)$ holds for all $m < n$. Why is this proof principle sound? Because there is no infinitely descending chain $m_0 > m_1 > \cdots$ of natural numbers. The principle of **well-founded induction** is a generalization of induction from $(\mathbb{N}, >)$ to any terminating reduction system $(A, \to)$. Formally, it is expressed by the following inference rule:

$$\frac{\forall x \in A.\ (\forall y \in A.\ x \xrightarrow{+} y \ \Rightarrow\ P(y)) \ \Rightarrow\ P(x)}{\forall x \in A.\ P(x)} \quad \text{(WFI)}$$

where P is some property of elements of A. The horizontal line is simply another symbol for implication.

In words: to prove $P(x)$ for all x, it suffices to prove $P(x)$ under the assumption that $P(y)$ holds for all successors y of x.

It may come as a bit of a surprise to see an induction schema without explicit base case. The solution to this puzzle is that the premise of WFI subsumes the base case. If $\rightarrow$ is terminating, the "base case" of the induction consists of showing that $P(x)$ holds for all elements without successor, i.e. all normal forms. Hence the assumption $(\forall y \in A.\ x \xrightarrow{+} y \ \Rightarrow\ P(y))$ is trivially true and the premise of WFI degenerates to $P(x)$, just as expected.

WFI is not correct for arbitrary $\rightarrow$, but for terminating ones it is:

Theorem 2.2.1 *If $\rightarrow$ terminates then WFI holds.*

Proof by contraposition. Assume that WFI does not hold for $\rightarrow$, i.e. there is some P such that the premise of WFI holds but the conclusion does not, i.e. $\neg P(a_0)$ for some $a_0 \in A$. But then the premise of WFI implies that there must exist some a_1 such that $a_0 \xrightarrow{+} a_1$ and $\neg P(a_1)$. By the same argument, there must exist some a_2 such that $a_1 \xrightarrow{+} a_2$ and $\neg P(a_2)$. Hence there is an infinite chain $a_0 \xrightarrow{+} a_1 \xrightarrow{+} a_2 \xrightarrow{+} \cdots$, i.e. $\rightarrow$ does not terminate. □

As a first application of WFI, we can prove the converse of this theorem:

Theorem 2.2.2 *If WFI holds, then $\rightarrow$ terminates.*

Proof by WFI where $P(x) :=$ "there is no infinite chain starting from x". The induction step is simple: if there is no infinite chain starting from any successor of x, then there is no infinite chain starting from x either. Hence the premise of WFI holds and we can conclude that $P(x)$ holds for all x, i.e. $\rightarrow$ terminates. □

A few words on terminology. Terminating relations are usually called **well-founded** in the mathematical literature. Hence the term "well-founded induction". In the computer science literature the terms **Noetherian**† and **Noetherian induction** are sometimes used instead. Strictly speaking, a reduction system $(A, \rightarrow)$ is well-founded if every nonempty $B \subseteq A$ has a minimal element, i.e. some $b \in B$ such that $b \rightarrow b'$ for no $b' \in B$. With

† In honour of the mathematician Emmy Noether.

the help of the Axiom of Choice it can be shown that well-foundedness and termination are equivalent.

We will now use well-founded induction to study some further properties of reductions which are related to termination.

Definition 2.2.3 A relation $\to$ is called

finitely branching if each element has only finitely many direct successors,
globally finite if each element has only finitely many successors,
acyclic if there is no element a such that $a \xrightarrow{+} a$.

Note that $\to$ is globally finite iff $\xrightarrow{+}$ is finitely branching.

Lemma 2.2.4 *A finitely branching relation is globally finite if it is terminating.*

Proof Let $\to$ be finitely branching and terminating. We use well-founded induction to prove that for every element the set of all its successors is finite. Since this is true for all its direct successors (by induction hypothesis), of which there are only finitely many, it is also true for the element itself. □

It is not true that a finitely branching relation is terminating if it is globally finite. The reason is cycles. However, we have the following weaker implication:

Lemma 2.2.5 *Any acyclic relation is terminating if it is globally finite.*

The combination of the last two lemmas says that a finitely branching and acyclic relation is globally finite iff it is terminating. The special case of an acyclic relation induced by a tree is known as **König's Lemma**:

A finitely branching tree is infinite iff it contains an infinite path.

Exercises

2.4 Show that $\xrightarrow{+}$ is terminating iff $\to$ is.

2.5 Show that $\xrightarrow{+}$ is a strict partial order iff $\to$ is acyclic.

2.6 A relation $\to$ is called **bounded** iff for each element the length of all paths starting from it is bounded: $\forall x.\ \exists n.\ \nexists y.\ x \xrightarrow{n} y$.

(a) Is every terminating relation bounded?

(b) Show that a finitely branching relation terminates iff it is bounded.

2.7 Prove Lemma 2.2.5.

2.3 Proving termination

The importance of termination hardly needs emphasizing: it is essential not just for programmers but also for theoreticians, as the previous sections, in particular the connection with well-founded induction, have shown. We will now examine a number of constructions for proving termination, a hard (because undecidable) task, as computer scientists well know. These constructions are on the level of relations and are applicable to termination proofs of programs as well as to purely mathematical questions, for example from the realm of group theory.

In connection with termination, it frequently pays to work with transitive relations or even partial orders. One reason is that there is a vast body of mathematical literature on partial orders. Another is that some of our constructions (e.g. the multiset order) are simpler for partial orders than for arbitrary relations. Fortunately, the transition to partial orders is without loss of generality: $\overset{+}{\to}$ terminates iff $\to$ does, in which case $\overset{+}{\to}$ is a strict order (Exercises 2.4 and 2.5).

The most basic method for proving termination of some $(A, \to)$ is to embed it into another abstract reduction system $(B, >)$ which is known to terminate. This requires a monotone mapping $\varphi : A \to B$, where **monotone** means that $x \to x'$ implies $\varphi(x) > \varphi(x')$. Now $\to$ terminates because an infinite chain $x_0 \to x_1 \to \cdots$ would induce an infinite chain $\varphi(x_0) > \varphi(x_1) > \cdots$. The mapping φ is often called a **measure function** and the whole construction is known as the **inverse image** construction (because $\to \subseteq \varphi^{-1}(>) := \{(x, x') \mid \varphi(x) > \varphi(x')\}$). Note that if φ is the identity, this yields that any subset of a terminating relation is terminating.

Example 2.3.1 The most popular choice for termination proofs is an embedding into $(\mathbb{N}, >)$, which is known to terminate. For strings, i.e. $A := X^*$ for some set X, there are two natural choices:

1. Length. φ is defined by $\varphi(w) := |w|$. This proves termination of all length-decreasing reductions like $uabbv \to_1 uaav$, where $u, v \in A$ are arbitrary and $a, b \in X$ are fixed.
2. Letters. For each $a \in X$ define $\varphi_a(w) :=$ "the number of occurrences of a in w". This can cope with reductions like $uav \to_2 vbu$ where $u, v \in A$ are arbitrary and $a, b \in X$, $a \neq b$, are fixed.

How about $\to_1 \cup \to_2$? We claim it also terminates, in which case Lemma 2.3.3 below tells us that there exists a measure function into $\mathbb{N}$. Can you find one?

Many program termination proofs follow the same schema by showing that every computation step (e.g. loop iteration or recursive call) decreases the value of some expression $\varphi(\overline{x})$ in terms of the program variables $\overline{x}$.

Example 2.3.2 Assume all variables in the following program range over natural numbers:

```
while $ub > lb+1$ do
begin $r$ := $(ub+lb)$ $div$ 2;
      if $\Phi$ then $ub$ := $r$ else $lb$ := $r$
end
```

Termination is independent of the test Φ (provided Φ terminates and has no side effect) and can be proved with the measure function $\varphi(ub, lb) := ub - lb$ which decreases with every loop iteration.

The popularity of measure functions into $\mathbb{N}$ is in part explained by the following completeness result:

Lemma 2.3.3 *A finitely branching reduction terminates iff there is a monotone embedding into* $(\mathbb{N}, >)$.

Proof The $\Leftarrow$-direction follows from the soundness of the measure function approach. For the other direction, let $\to$ be a terminating and finitely branching reduction. Define $\varphi(x)$ as the number of successors of x which, by Lemma 2.2.4, must be finite. Since $\to$ is terminating and hence acyclic, $x \to x'$ implies that x' has strictly fewer successors than x. Alternatively, $\varphi(x)$ can be defined as the length of the longest reduction starting from x. Since $\to$ terminates, Exercise 2.6 implies that $\varphi(x)$ is well-defined. □

The restriction to finitely branching relations is necessary, as the following example shows.

Example 2.3.4 Let $A := \mathbb{N} \times \mathbb{N}$ and let $\to$ be defined by the two rules $(i+1, j) \to (i, k)$ and $(i, j+1) \to (i, j)$ for all $i, j, k \geq 0$. This reduction is not finitely branching because the value of k in the first rule is not constrained by the left-hand side. Termination of $\to$ can be shown by a simple lexicographic construction (see Section 2.4). Yet there is no monotone function φ from $(\mathbb{N} \times \mathbb{N}, \to)$ into $(\mathbb{N}, >)$. For if there were such a function φ, observe that monotonicity implies $k := \varphi(1,1) > \varphi(0,k) > \varphi(0,k-1) > \cdots > \varphi(0,0)$. This is a contradiction because there are only k natural numbers below k and yet the chain $\varphi(0,k) > \cdots > \varphi(0,0)$ has length $k+1$.

Even in the context of finitely branching reductions, an embedding into $\mathbb{N}$ can be tricky to find.

Example 2.3.5 Let $A = \mathbb{N}\times\mathbb{N}$ and define the reduction by $(i, j+1) \to (i, j)$ and $(i+1, j) \to (i, i)$. This reduction terminates at $(0, 0)$ for every start point. It is also finitely branching. Hence there is a measure function into $\mathbb{N}$. In this particular case $\varphi(i, j) = i^2+j$ does the job, but it takes a moment to find this function and prove that it is monotone.

We will now discuss how to get around the above problems with measure functions into $\mathbb{N}$ by building complex orders from simpler ones using fixed constructions which preserve termination.

Exercises

2.8 Find a measure function into $\mathbb{N}$ which proves termination of $\to$ in Example 2.1.2, part 2.

2.9 Find a measure function into $\mathbb{N}$ which proves termination of $\to_1 \cup \to_2$ in Example 2.3.1.

2.4 Lexicographic orders

Given two strict orders $(A, >_A)$ and $(B, >_B)$, the **lexicographic product** $>_{A\times B}$ on $A \times B$ is defined by

$$(x, y) >_{A\times B} (x', y') \;:\Leftrightarrow\; (x >_A x') \vee (x = x' \wedge y >_B y').$$

If A and B are obvious from the context we write $>$ instead of $>_{A\times B}$. Sometimes we also write $>_A \times_{lex} >_B$.

The following property is routine to prove:

Lemma 2.4.1 *The lexicographic product of two strict orders is again a strict order.*

More interestingly we have

Theorem 2.4.2 *The lexicographic product of two terminating relations is again terminating.*

Proof by contradiction. Assume there is an infinitely descending chain $(a_0, b_0) > (a_1, b_1) > \cdots$. This implies $a_0 \geq_A a_1 \geq_A \cdots$. Since $>_A$ terminates, this chain cannot contain an infinite number of strict steps $a_i >_A a_{i+1}$. Hence there is a k such that $a_i = a_{i+1}$ for all $i \geq k$. But this implies $b_i >_B b_{i+1}$ for all $i \geq k$, which contradicts the termination of $>_B$. □

This theorem proves termination of $\to$ on $\mathbb{N} \times \mathbb{N}$ in Examples 2.3.4 and 2.3.5: $(i, j) \to (i', j')$ is defined such that (i, j) is lexicographically greater than (i', j'), i.e. $\to$ is a subset of the terminating relation $>_{\mathbb{N}\times\mathbb{N}}$. It also proves termination of $\to_1 \cup \to_2$ in Example 2.3.1: $\to_1$ decreases the length whereas $\to_2$ leaves the length invariant but decreases the number of as.

Lexicographic products are essential in building up more complex orders from simpler ones. By iteration, we can form lexicographic products over any number of strict orders $(A_i, >_i)$, $i = 1, \ldots, n$: $>_{1\ldots n}$, where $n > 1$, is the lexicographic product of $>_1$ and $>_{2\ldots n}$. Unwinding the recursion and writing $>$ instead of $>_{1\ldots n}$ we get

$$(x_1, \ldots, x_n) > (y_1, \ldots, y_n) \;:\Leftrightarrow\; \exists k \leq n.\ (\forall i < k.\ x_i = y_i) \wedge x_k >_k y_k. \quad (2.1)$$

If all $(A_i, >_i)$ are the same we write $>^n_{lex}$ for the n-fold lexicographic product.

The above results for the binary lexicographic product carry over to n-fold products: $>$ is again a strict order and it terminates if all the $>_i$ terminate. The proofs are by induction on n.

Instead of tuples of fixed length, we can also consider strings of arbitrary but finite length: given a strict order $(A, >)$, the **lexicographic order** $>^*_{lex}$ on A^* is defined as

$$u >^*_{lex} v \;:\Leftrightarrow\; (|u| > |v|) \vee (|u| = |v| \wedge u >^{|u|}_{lex} v)$$

where $|w|$ is the length of w and $>^{|w|}_{lex}$ is the order on $A^{|w|}$ defined in (2.1) above. More concisely, we can define $>^*_{lex}$ as the lexicographic product of $>_{len}$ and $\bigcup_{i\in\mathbb{N}} >^i_{lex}$, where $u >_{len} v :\Leftrightarrow |u| > |v|$. Since A^i and A^j are disjoint if $i \neq j$, the second component of this product is a union of orders over disjoint sets. Since such unions (this is easy to see) and lexicographic products (as shown above) preserve strict orders and termination, we have

Lemma 2.4.3 *If $>$ is a strict order, so is $>^*_{lex}$. If $>$ terminates, so does $>^*_{lex}$.*

Despite its name, $>^*_{lex}$ is not the order used in dictionaries. The latter does not terminate: $a >_{dict} aa >_{dict} aaa >_{dict} \cdots$.

Yet another interesting variation on lexicographic orders compares strings from left to right as follows: $w_1 >_{Lex} w_2$ if w_2 is a proper prefix of w_1 or if $w_1 = uav$, $w_2 = ubw$ and $a > b$, where $>$ is the underlying strict order. For example, if $a > b$, then $aaaa >_{Lex} aaa >_{Lex} abba$. Unfortunately, $>_{Lex}$ need not terminate either, even if $>$ does (exercise!). Nevertheless, $>_{Lex}$ can be a useful component in more complicated orders.

Lemma 2.4.4 *If $>$ is a strict order, so is $>_{Lex}$.*

The proof, a simple case analysis, is left as an exercise.