Cognitive Technologies

Managing Editors: D.M. Gabbay J. Siekmann

Editorial Board: A. Bundy J.G. Carbonell
M. Pinkal H. Uszkoreit M. Veloso W. Wahlster
M. J. Wooldridge

Springer-Verlag Berlin Heidelberg GmbH

J. W. Lloyd

Logic for Learning

Learning Comprehensible Theories from Structured Data

Springer

J. W. Lloyd
The Australian National University
Research School of Information Sciences
and Engineering
Computer Sciences Laboratory
Canberra ACT 0200
Australia
jwl@csl.anu.edu.au

With 14 Figures

Cataloging-in-Publication Data applied for

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche
Nationalbibliografie; detailed bibliographic data is available in the
Internet at <http://dnb.ddb.de>.

ACM Computing Classification (1998): I.2.6, I.2.4, I.2.3
ISSN 1611-2482
ISBN 978-3-642-07553-7 ISBN 978-3-662-08406-9 (eBook)
DOI 10.1007/978-3-662-08406-9

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German copyright law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag Berlin Heidelberg GmbH. Violations are liable for prosecution under the German Copyright Law.

http://www.springer.de

© J. W. Lloyd 2003
Originally published by Springer-Verlag Berlin Heidelberg New York in 2003
Softcover reprint of the hardcover 1st edition 2003
The use of general descriptive names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera-ready by the author
Printed on acid-free paper 45/3142SR - 5 4 3 2 1 0

To
Susan, Simon, and Patrick

Preface

This book is concerned with the rich and fruitful interplay between the fields of computational logic and machine learning. The intended audience is senior undergraduates, graduate students, and researchers in either of those fields. For those in computational logic, no previous knowledge of machine learning is assumed and, for those in machine learning, no previous knowledge of computational logic is assumed.

The logic used throughout the book is a higher-order one. Higher-order logic is already heavily used in some parts of computer science, for example, theoretical computer science, functional programming, and hardware verification, mainly because of its great expressive power. Similar motivations apply here as well: higher-order functions can have other functions as arguments and this capability can be exploited to provide abstractions for knowledge representation, methods for constructing predicates, and a foundation for logic-based computation.

The book should be of interest to researchers in machine learning, especially those who study learning methods for structured data. Machine learning applications are becoming increasingly concerned with applications for which the individuals that are the subject of learning have complex structure. Typical applications include text learning for the World Wide Web and bioinformatics. Traditional methods for such applications usually involve the extraction of features to reduce the problem to one of attribute-value learning. The book investigates alternative approaches that involve learning directly from an accurate representation of the complex individuals and provides a suitable knowledge representation formalism and generalised learning algorithms for this purpose. Throughout, great emphasis is placed on learning *comprehensible* theories. There is no attempt at a comprehensive account of machine learning; instead the book concentrates largely on the problem of learning from structured data. For those readers primarily interested in the applications to machine learning, a 'shortest path' through the preceding chapters to get to this material is indicated in Chap. 1.

The book serves as an introduction for computational logicians to machine learning, a particularly interesting and important application area of logic, and also provides a foundation for functional logic programming languages. However, it does not provide a comprehensive account of higher-order

logic, much less computational logic, concentrating instead on those aspects of higher-order logic that can be applied to learning.

There is also something new here for researchers in knowledge representation. The requirement of a suitable formalism for representing individuals in machine-learning applications has led to the development of a novel class of higher-order terms for representing such individuals. The theoretical foundations of this class of terms are developed in detail. While the main application here is to machine learning, this class has applications throughout computer science wherever logic is used as a knowledge representation formalism.

There is a Web site for the book that can be found at
`http://discus.anu.edu.au/~jwl/LogicforLearning/`.
The ALKEMY learning system described in the book can be obtained from there.

I am greatly indebted to my collaborators Antony Bowers, Peter Flach, Thomas Gärtner, Christophe Giraud-Carrier, and Kee Siong Ng over the last six years. Without their contributions, this book would have not been possible. Kee Siong Ng implemented ALKEMY and contributed greatly to its design. I thank Hendrik Blockeel, Luc De Raedt, Michael Hanus, Stefan Kramer, Nada Lavrač, Stephen Muggleton, David Page, Ross Quinlan, Claude Sammut, Jörg Siekmann, and Ashwin Srinivasan for technical advice on various aspects of the material. Finally, this book builds upon a long tradition of logical methods in machine learning. In this respect, I would like to acknowledge the works of Gordon Plotkin, Ryszard Michalski, Ehud Shapiro, Ross Quinlan, Stephen Muggleton, and Luc De Raedt that have been particularly influential.

Canberra, May 2003 *John Lloyd*

Contents

1. Introduction 1
1.1 Outline of the Book 1
1.2 Setting the Scene 5
1.3 Introduction to Learning 10
1.4 Introduction to Logic 16
Bibliographical Notes 27
Exercises 28

2. Logic 31
2.1 Types 31
2.2 Type Substitutions 35
2.3 Terms 38
2.4 Subterms 45
2.5 Term Substitutions 55
2.6 λ-Conversion 64
2.7 Model Theory 72
2.8 Proof Theory 76
Bibliographical Notes 79
Exercises 80

3. Individuals 83
3.1 Default Terms 83
3.2 Normal Terms 89
3.3 An Equivalence Relation on Normal Terms 93
3.4 A Total Order on Normal Terms 95
3.5 Basic Terms 97
3.6 Metrics on Basic Terms 105
3.7 Kernels on Basic Terms 115
Bibliographical Notes 127
Exercises 128

4. Predicates 131
4.1 Transformations 131
4.2 Standard Predicates 139

4.3 Regular Predicates 146
4.4 Predicate Rewrite Systems 151
4.5 The Implication Preorder 158
4.6 Efficient Construction of Predicates 163
Bibliographical Notes 175
Exercises 176

5. Computation 183
5.1 Programs as Equational Theories 183
5.2 Definitions of Some Basic Functions 188
5.3 Programming with Abstractions 193
Bibliographical Notes 203
Exercises 203

6. Learning 207
6.1 Decision-Tree Learning 207
6.2 Illustrations 214
Bibliographical Notes 240
Exercises 241

A. Appendix 243
A.1 Well-Founded Sets 243

References 245

Notation 251

Index 253

1. Introduction

After an outline of the book, this chapter gives a brief historical introduction to computational logic and machine learning, and their intersection. It also provides some motivation for the topics studied in the form of introductions to learning and to logic.

1.1 Outline of the Book

This book is concerned with the interplay between logic and learning. It consists of six chapters.

This chapter provides an overview of higher-order logic and its application to learning. It is written in an informal manner. No previous knowledge of computational logic or machine learning is assumed. Thus the section that introduces learning does so assuming the reader has no previous knowledge of that field. Similarly, the section that introduces logic explains the main ideas of logic, and especially the ones that are emphasised in this book, from the beginning. Preceding these sections is one that provides an historical context for the material that follows.

Chapter 2 is concerned with the detailed development of the logic itself. The logic employed is a higher-order one because this most naturally provides the concepts needed in applications. It is based on the classical higher-order logic of Church introduced in his simple theory of types, which is referred to as type theory in the following. In fact, the logic presented here extends type theory in that it is polymorphic and admits product types. The polymorphism introduced is a simple form of parametric polymorphism. A declaration for a polymorphic constant is understood as standing for a collection of declarations for the (monomorphic) constants that can be obtained by instantiating all parameters in the polymorphic declaration with closed types. Similarly, a polymorphic term can be regarded as standing for a collection of (monomorphic) terms. The development of the logic is mainly focussed on those topics that are needed to support its application to machine learning.

In Chap. 3, a set of terms that is suitable for representing individuals in diverse applications is identified. The most interesting aspect of this set of what are called basic terms is that it includes certain abstractions and therefore is larger than is normally considered for knowledge representation.

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

These abstractions allow one to model sets, multisets, and data of similar types, in a direct way. This chapter also shows how to construct metrics and kernels on basic terms; these are needed for metric-based and kernel-based learning methods.

In Chap. 4, a systematic method for constructing predicates on individuals is presented. For this purpose, particular kinds of functions, called transformations, are defined and predicates are constructed incrementally by composing transformations. Each hypothesis language is specified by a predicate rewrite system that determines those predicates that are to be admitted. Predicate rewrite systems give users precise and explicit control over the hypothesis language.

Chapter 5 provides a computational framework for a variety of applications, including machine learning. The approach taken here is that a declarative program is an equational theory and that computation is simplification of terms by rewriting. Thus another difference compared with the original formulation of type theory is that the proof theory developed by Church (and others) is modified here to give a more direct form of equational reasoning that is better suited to the application of the logic as a foundation for declarative programming languages. Of particular interest is that redexes can contain abstractions. This approach allows a uniform treatment of set and multiset processing, as well as processing of the quantifiers.

In Chap. 6, the material developed in the book is applied to the problem of learning comprehensible theories from structured data. The general approach to learning involves recursive partitioning of the set of training examples by well-chosen predicates. The resulting theories are essentially decision trees that generally can be easily comprehended. Chapter 3 provides the knowledge representation formalism for the individuals, Chap. 4 the method of predicate construction for partitioning the training examples, and Chap. 5 the computational model for evaluating predicates applied to individuals for the learning process. Chapter 6 also contains a description of the ALKEMY decision-tree learning system which is applied to a diverse set of learning applications that illustrate the ideas introduced in the book.

An appendix provides some material on well-founded sets.

Each chapter has a series of exercises of varying difficulty. Some open research problems are also given. Each chapter has bibliographical notes to provide pointers to the original sources of results and related material.

The methods introduced here to address the problem of learning from structured data have wide applicability throughout machine learning, beyond the particular focus on logical methods and comprehensibility of this book. The reason for their wide applicability is that increasingly one has to deal with learning problems for which the individuals that are the subject of learning have complex structure. Such applications abound, for example, in bioinformatics and text learning. The traditional approach usually involves representing the individuals using the attribute-value language (that is, by a

vector of numbers and/or constants). In contrast, the approach in this book involves directly representing the structure of the individuals by basic terms. Once this representation has been satisfactorily carried out, one then has the choice of either using learning methods suitably generalised to directly handle basic terms, as studied in this book, or else using the accurate representation as a basis for feature extraction after which conventional learning methods can be used. Whatever learning methods are ultimately applied, the first stage of accurately representing the individuals in a suitably rich knowledge representation language is important. For example, the detailed type information in the representation strongly suggests the conditions that could be used to split sets of labelled individuals and this provides the basis for decision-tree learning algorithms. Furthermore, even if one wants to extract features at an early stage, it is crucial to know whether the individuals under consideration are lists or sets, for example, as the likely features differ greatly for each of these two cases.

The issue of comprehensibility in learning also pervades the book and contrasts with many other learning techniques, such as neural networks and support vector machines, that do not provide comprehensible theories. Thus the book is partly about scientific discovery – one wants to be able to show straightforwardly why a particular theory *explains* some observations.

This book should be of interest to researchers in computational logic who do not know machine learning. As the many interesting and important applications show, machine learning is an indispensable technology for achieving the aims of artificial intelligence. For complex machine-learning applications, logic provides a convenient and effective knowledge representation and computational formalism. This book is a suitable vehicle for introducing computational logicians to this exciting application area. Even for readers with no interest in machine learning, the book provides a foundation for higher-order computational logic that should be of interest to those who work in functional logic programming, knowledge representation, and other parts of computational logic.

The range of learning problems considered in this book is essentially the same as that of inductive logic programming (ILP), a subfield of machine learning concerned with the application of first-order logic to learning problems. However, while the starting point may have been ILP, the presentation provided here differs from that approach, since it has resulted from a fresh look at the foundations of ILP. In particular, the presentation here draws upon the experience gained from working on the problem of integrating functional and logic programming languages and is motivated by the attractiveness of the typed, higher-order approach of a typical functional programming language, such as Haskell. With this background, it is natural to try to reconstruct ILP in a typed, higher-order context.

In this reconstruction, the first key idea is that individuals should be represented by terms. For this idea to work, it is essential that sets, multisets,

and similar terms be available. In higher-order logic, a set is identified with its characteristic function, that is, a set is a predicate. Similarly, a multiset is a mapping into the natural numbers. Certain abstractions are then used to represent sets, multisets, and data of similar types. Higher-order logic also provides all the machinery needed to process terms of these types.

Having represented the individuals as terms, one is then faced with the actual learning problem: how should one learn some classification function, for example, defined over the individuals. The key idea here, especially if one wants to induce comprehensible theories, is to find conditions that separate the training examples into (sufficiently) pure subsets, where ‘pure’ means belonging to the same class. Thus one is led to the problem of finding predicates that can be used to partition the training examples. The higher-order nature of the logic can be exploited once again: predicates are constructed by composing transformations appropriate to the application. Precise control is exercised over the hypothesis space by specifying a system of rewrites that is used to generate predicates. Thus the higher-order nature of the logic has been used in two essential ways, by providing abstractions for representing individuals and by providing composition for the construction of predicates.

Higher-order logic is undecidable in several respects: unification of terms and checking a formula for theoremhood are both undecidable. But unification (of higher-order terms) is not needed for the applications to either declarative programming languages or machine learning. Also successful programming languages such as Haskell and λProlog show that subsets of the logic can be used efficiently. Furthermore, the use of an expressive formalism like higher-order logic in machine learning does not somehow make the learning problem harder or more complex. In fact, if anything, the reverse is true, since the richer knowledge representation language provides a direct representation of individuals and a perspicuous approach to predicate construction. Furthermore, the complexity is in the learning problem itself, not the knowledge representation formalism used to solve it, especially if the formalism provides direct representations of individuals and predicates, as is true of the approach here.

Finally, I emphasise that a lack of knowledge of higher-order logic, even logic itself, should not be a deterrent from reading this book as all the logic that is needed for learning is provided here.

Shortest Path to the Machine Learning Applications

To help those readers who are primarily interested in the applications to machine learning and who would prefer to learn just enough logic to understand those applications, I indicate a ‘shortest path’ through Chaps. 1 to 5 to get to the material on learning in Chap. 6.

First, it is necessary to read all of the present chapter as it gives an informal account of the material that follows. Then the following sections from Chaps. 2 to 4 should be read: 2.1, 2.2, 2.3, 2.4 (first two pages), 2.5

(first two pages), 3.1, 3.2, 4.1, 4.2, 4.3, and 4.4. It would even be possible to omit all the proofs in these sections at a first reading. As a guide, the key concepts to look for are type, type substitution, term, subterm, type weaker, term substitution, normal term, transformation, standard predicate, regular predicate, and predicate rewrite system. The representation of individuals actually uses basic terms from Sect. 3.5, but substituting normal terms from Sect. 3.2 suffices at a first reading to understand the material in Chap. 6.

Chapter 5 can be omitted at a first reading as the intuitive understanding of computation given in the present chapter suffices to understand Chap. 6.

In addition, readers interested in metric-based learning should read Sect. 3.6 and those in kernel-based learning should read Sect. 3.7.

The book contains a large number of rather technical results and, even for a reader who intends to go through the entire book in detail, it is helpful to establish in advance which of these results are the most important.

In Chap. 2, Propositions 2.5.2 and 2.5.4, which are concerned with whether a substitution applied to a term produces a term again, are heavily used throughout. On a similar theme, Proposition 2.4.6, is concerned with a particular situation in which replacing a subterm of a term by another term gives a term again. Part 2 of Proposition 2.6.4 establishes an important property of β-reduction. Proposition 2.8.2 is a technical result that is used to establish important properties of proofs and computations. The soundness of the proof theory is given by Proposition 2.8.3.

In Chap. 3, Proposition 3.6.1 establishes a metric and Proposition 3.7.1 a kernel on sets of basic terms.

In Chap. 4, Propositions 4.5.3 and 4.6.10 provide important properties of predicate rewrite systems.

Chapter 5 contains two main results. Proposition 5.1.3 shows that run-time type checking is unnecessary and Proposition 5.1.6 establishes an important correctness property of computations.

Many proofs use structural induction because of the inductive definition of the concepts of interest.

1.2 Setting the Scene

This section contains brief historical sketches of the fields of computational logic and machine learning, and their intersection.

Computational Logic

Logic, of which computational logic is a subfield, is one of the oldest and richest scientific endeavours, going back to the ancient Greeks. The original motivation was to understand and formalise reasoning and this drove the philosophical investigations into logic, by Aristotle, Hobbes, Leibniz, and

Boole, for example. A landmark contribution was that of Frege in 1879 when his *Begriffsschrift* – meaning something like 'concept writing' – was published. While Frege's notation is unlike anything we use today, *Begriffsschrift* is essentially what is now known as first-order logic. Over the next few decades, set theory, axiomatised in first-order logic, was employed as the foundation of mathematics, although it had to survive various traumas such as the one initiated by Russell's paradox. Later, in 1931, Gödel powerfully demonstrated the limitations of the axiomatic approach with his incompleteness theorem. A more recent relevant development was Church's simple theory of types, introduced in 1940, which was partly motivated by the desire to give a typed, higher-order foundation to mathematics.

Around 1957, computers were sufficiently widespread and powerful to encourage researchers to attempt an age-old dream of philosophers – the automation of reasoning. After some early attempts by Gilmore, Wang, Davis, Putnam, and others, to build automatic theorem provers, Robinson introduced the resolution principle in 1963. This work had an extraordinary impact and led to a flowering of research into theorem proving, so that today many artificial intelligence systems have a theorem prover at their heart. In the last four decades, computational logic, understood broadly as the use of logic in computer science, has developed into a rich and fruitful field of computer science with many interconnected subfields.

I turn now more specifically to higher-order logic. The advantages of using a higher-order approach to computational logic have been advocated for at least the last 30 years. First, the functional programming community has used higher-order functions from the very beginning. The latest versions of functional languages, such as Haskell98, show the power and elegance of higher-order functions, as well as related features such as strong type systems. Of course, the traditional foundation for functional programming languages has been the λ-calculus, rather than a higher-order *logic*. However, it is possible to regard functional programs as equational theories in a logic such as the one introduced here and this also provides a useful semantics.

In the 1980s, higher-order programming in the logic programming community was introduced through the language λProlog. The logical foundations of λProlog are provided by almost exactly the logic studied in this book. However, a different sublogic is used for λProlog programs than the equational theories proposed here. In λProlog, program statements are higher-order hereditary Harrop formulas, a generalisation of the definite clauses used by Prolog. The language provides an elegant use of λ-terms as data structures, meta-programming facilities, universal quantification and implications in goals, amongst other features.

A long-term interest amongst researchers in declarative programming has been the goal of building integrated functional logic programming languages. Probably the best developed of these functional logic languages is the Curry language, which is the result of an international collaboration over the last

decade. To quote from the Curry report: "Curry is a universal programming language aiming to amalgamate the most important declarative programming paradigms, namely functional programming and logic programming. Moreover, it also covers the most important operational principles developed in the area of integrated functional logic languages: 'residuation' and 'narrowing'. Curry combines in a seamless way features from functional programming (nested expressions, higher-order functions, lazy evaluation), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of expressions with synchronisation on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic and demand-driven evaluation of functions)."

There are many other outstanding examples of systems that exploit the power of higher-order logic. For example, the HOL system is an environment for interactive theorem proving in higher-order logic. Its most outstanding feature is its high degree of programmability through the meta-language ML. The system has a wide variety of uses from formalising pure mathematics to verification of industrial hardware. In addition, there are at least a dozen other systems related to HOL. On the theoretical side, much of the research in theoretical computer science, especially semantics, is based on the λ-calculus and hence is intrinsically higher order in nature.

Machine Learning

Now I turn to machine learning, which has had a similarly rich, although very different, history. The motivating goal in machine learning is to build computer systems that can improve their performance according to their experience. There is good reason to want such systems: as we attempt to build more and more complex computer systems, it becomes increasingly difficult to plan for all the likely situations that the systems will meet in their lifetimes. Thus it makes sense to design and implement architectures that are flexible enough to allow computer systems to adapt their behaviour according to the circumstances.

What is most striking about machine learning is that so many other disciplines have contributed substantially to it, and continue to do so. Indeed, many problems of machine learning were studied in these disciplines before machine learning came to be recognised as an independent field in the 1960s. In no particular order, here are the main contributing disciplines.

Statisticians have long been concerned with the general problem of extracting patterns and trends from (possibly very large amounts of) data and thus explaining what the data 'means'. Typical problems include predicting whether a patient, having had one heart attack, is likely to have another and estimating the risk factors for various kinds of cancer. These problems are

also typical machine-learning problems, so it is not so surprising that some learning methods (decision-tree learners, for example) were developed independently and simultaneously in both fields. More recently, the interactions between machine-learning researchers and statisticians have been much more synergistic.

Engineers have long been concerned with the problem of control. There is a very close connection between adaptive control theory and what is called reinforcement learning in machine learning. Reinforcement learning is concerned with the problem of how an agent receiving percepts from an environment can learn to perform actions that will allow it to achieve its aim(s). A typical application is training a robot to successfully negotiate the corridors of a building. The training method involves from time to time giving the agent rewards that are positive (if the agent has performed well) and negative (if the agent has performed badly). Over a series of training exercises, the agent has to learn from these (delayed) rewards a policy that tells it how to act as optimally as possible according to its perceived state of the environment. This approach relies heavily on research in dynamic programming and Markov decision processes developed by engineers in the 1960s.

An important input to machine learning has come from physiologists and psychologists who have attempted to model the human brain. The work of McCulloch and Pitts in 1943, Hebb in 1949, and Rosenblatt in 1958 on various kinds of neuron models led eventually, after a period of stagnation, to the resurgence around 1986 of what are now usually called artificial neural networks. These are networks of interconnected units, where the units are mathematical idealisations of a single neuron. Like neurons, the units fire if their input exceeds some threshold. The resurgence of these ideas in the 1980s came about because of the discovery that neural networks could be effectively trained by a simple iterative algorithm, called backpropagation. Today, neural networks are one of the most commonly used learning methods.

Another biologically inspired input is that of genetic algorithms that are based loosely on evolution. In this approach, hypotheses are usually described by bit strings (which correspond to the DNA of some species). Then the learning process involves searching for a suitable hypothesis by starting with some initial population of hypotheses and applying the operations of mutation and crossover (which mimics sexual reproduction) to form subsequent populations. At each step, the current hypotheses are evaluated by a fitness function with the most fit hypotheses being selected for the next generation. Genetic algorithms have been successfully applied to a variety of learning and optimisation problems.

Finally, there is the influence from artificial intelligence, which from its early days around 1956 provided researchers with a strong motivation to build programs that could learn. An outstanding early example was the checker-playing program of Samuels in 1959 that employed ideas similar to reinforce-

ment learning. In the 1950s, CLS (Concept Learning System) was developed by Hunt. This was based on the idea of recursive partitioning of the training examples and was highly influential in development of the decision-tree systems that followed. An early motivation for this kind of learning system was the desire to automate the knowledge acquisition task for building expert systems, as explicitly expressed by Quinlan in 1979, who developed the ID3 system and later the C4.5 and C5.0 systems that are widely used today. Another important early development was the version space concept of Mitchell whereby the general-to-specific ordering of hypotheses was exploited to efficiently search the hypothesis space.

Logic and Learning

So far it is not apparent where the connections between computational logic and machine learning lie. For this, one has to go back once again to the early philosophers. As well as studying the problem of deduction in logic (that is, what follows from what), philosophers were also interested in the problem of induction (that is, generalising from instances). Induction is fundamental to an understanding of the philosophy of science, since much of science involves discovering general laws by generalising from experimental data. Important contributions to the study of induction were made by Bacon, Mill, Jevons, and Peirce, for example.

With the availability of computers and the growth of artificial intelligence, the problem of induction and especially that of building inductive systems was studied by the pioneers of machine learning. An early use of first-order logic for knowledge representation in concept learning was published by Banerji in 1964. Then, in 1970, Plotkin formalised induction in (first-order) clausal logic. The motivation here was that, since unification (which finds the greatest common instance of a set of atoms) was the fundamental component of deduction, anti-unification (which finds the least common generalisation of a set of atoms) ought to be the key to induction. This work of Plotkin contains several seminal contributions including an anti-unification algorithm, the concept of relative subsumption (where 'relative' refers to a background theory), and a method of finding the relative least common generalisation of a set of clauses. Closely related work was done independently and contemporaneously by Reynolds.

From the early 1970s, Michalski studied inductive learning using various logical formalisms, and generalisation and specialisation rules. Other relevant work around this time includes that of Vere who in 1975 developed inductive algorithms in first-order logic, building on the earlier work of Plotkin. A little later, in 1981, Shapiro developed the influential model inference system that was the first learning system to explicitly make use of Horn clause logic, no doubt influenced by the arrival a few years earlier of the Prolog programming language. Amongst other contributions, he introduced the important idea of a refinement operator that is used to specialise a theory. The MARVIN system

of Sammut from 1981 was an interactive concept learner that employed both specialisation and generalisation. Buntine revived interest in subsumption as a method of generalisation for the context of Horn clause theories in 1986. Much of this earlier work was generalised in 1988 to the setting of inverse resolution by Buntine and Muggleton. Another influential system around this time was the FOIL system of Quinlan that induced Horn clause theories.

The increasing interest in logical formalisms for learning in the late 1980s led to the naming of the subfield of inductive logic programming by Muggleton and its definition as the intersection of logic programming and machine learning. The establishment of ILP as an independent subfield of machine learning was led by Muggleton and De Raedt. Much of the important work on learning in first-order logic has since taken place in ILP and most of the standard techniques of machine learning have now been upgraded to this context. This work is partly documented in the series of workshops on inductive logic programming that started in 1991 and continues to the present day.

1.3 Introduction to Learning

This section provides a tutorial introduction to the learning issues that will be of interest in this book.

An Illustration

Consider the problem of determining whether a bunch of keys, or more precisely some key on the bunch, can open a door. The data for this problem are a number of bunches of keys and the information about whether each bunch does or does not open the door. The problem is to find an hypothesis that agrees with the data that is given and, furthermore, will correctly predict whether new bunches of keys will open the door or not.

For this illustration, the individuals that are the subject of learning are the bunches of keys. First, these individuals have to be represented (that is, modelled in a suitable knowledge representation formalism). Now a bunch is nothing other than a set, so the problem reduces to representing a key. There are quite a number of choices in how to do this. Let us choose to represent a key by its values for four specific characteristics: its make, how many prongs it has, its length, and its width. Following standard methods of knowledge representation, this leads to the introduction of the four types:

$$Make, NumProngs, Length, \text{ and } Width.$$

Also required are some constants for each of these types. These are as follows.

$$Abloy, Chubb, Rubo, Yale : Make$$
$$Short, Medium, Long : Length$$
$$Narrow, Normal, Broad : Width.$$

The meaning of the first of these declarations is that *Make* is a type and *Abloy*, *Chubb*, *Rubo*, and *Yale* are constants of type *Make*. The other declarations have a similar meaning. The constants of type *NumProngs* are intended to be integers, so *NumProngs* is declared to be a synonym for *Int*, the type of the integers, by the declaration

$$NumProngs = Int.$$

Now a key is represented by a 4-tuple of constants from each of the four types. This is specified by the declaration

$$Key = Make \times NumProngs \times Length \times Width,$$

for which *Key* is the type of a key and $\times$ denotes (cartesian) product. Thus *Key* has been declared to be a synonym for the product type on the right-hand side, and 4-tuples, where the first component is a constant of type *Make*, the second is a constant of type *NumProngs* and so on, are used to represent keys. For example, the tuple

$$(Abloy, 3, Short, Normal)$$

represents the key whose make is Abloy, that has 3 prongs, is short, and has normal width.

A bunch of keys can now be represented as a set via the declaration

$$Bunch = \{Key\}.$$

This states that the type *Bunch* is a synonym for the type $\{Key\}$ which is the type of sets whose elements have type *Key*. A typical bunch is now represented by a set such as

$$\{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad), (Chubb, 3, Long, Narrow)\},$$

which is a bunch containing three keys. This completes the representation of the bunches of keys.

The next task is to make precise the type of the function that is to be learned. Recall that the illustration involved predicting whether or not a bunch of keys opened a particular door or not. This suggests that the function be a mapping from bunches of keys to the set of boolean values, true and false. Now the type of the booleans is denoted by Ω, and $\top$ is the constant representing true, and $\bot$ is the constant representing false. If the desired function is called *opens*, then it has the declaration

$$opens : Bunch \rightarrow \Omega.$$

The meaning of this declaration is that *opens* is a function from elements of type *Bunch* to elements of type Ω. The type $Bunch \rightarrow \Omega$ is the so-called

signature of the function. If one had a definition for *opens*, then, given a bunch, one could evaluate the function *opens* on the set representing this bunch to discover whether the bunch opens the door or not: if the function evaluated to $\top$, then the bunch would open the door; otherwise, it would not. The problem, of course, is to find a suitable definition for *opens*.

Part of the data for doing this are some examples that give the value of the function *opens* for some specific bunches of keys. This data is the so-called *training data*. Suppose that the examples are as follows.

$$
\begin{aligned}
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}), (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Chubb}, 3, \mathit{Long}, \mathit{Narrow})\} = \top\\
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Medium}, \mathit{Broad}), (\mathit{Chubb}, 2, \mathit{Long}, \mathit{Normal}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Chubb}, 4, \mathit{Medium}, \mathit{Broad})\} = \top\\
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Short}, \mathit{Broad}), (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Chubb}, 3, \mathit{Long}, \mathit{Narrow})\} = \top\\
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Medium}, \mathit{Broad}), (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Narrow}),\\
&\qquad\qquad (\mathit{Chubb}, 3, \mathit{Long}, \mathit{Broad}), (\mathit{Yale}, 4, \mathit{Medium}, \mathit{Broad})\} = \top\\
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Medium}, \mathit{Narrow}), (\mathit{Chubb}, 6, \mathit{Medium}, \mathit{Normal}),\\
&\qquad\qquad (\mathit{Rubo}, 5, \mathit{Short}, \mathit{Narrow}), (\mathit{Yale}, 4, \mathit{Long}, \mathit{Broad})\} = \top\\
&\mathit{opens}\ \{(\mathit{Chubb}, 3, \mathit{Short}, \mathit{Broad}), (\mathit{Chubb}, 4, \mathit{Medium}, \mathit{Broad}),\\
&\qquad\qquad (\mathit{Yale}, 3, \mathit{Short}, \mathit{Narrow}), (\mathit{Yale}, 4, \mathit{Long}, \mathit{Normal})\} = \bot\\
&\mathit{opens}\ \{(\mathit{Yale}, 3, \mathit{Long}, \mathit{Narrow}), (\mathit{Yale}, 4, \mathit{Long}, \mathit{Broad})\} = \bot\\
&\mathit{opens}\ \{(\mathit{Abloy}, 3, \mathit{Short}, \mathit{Broad}), (\mathit{Chubb}, 3, \mathit{Short}, \mathit{Broad}),\\
&\qquad\qquad (\mathit{Rubo}, 4, \mathit{Long}, \mathit{Broad}), (\mathit{Yale}, 4, \mathit{Long}, \mathit{Broad})\} = \bot\\
&\mathit{opens}\ \{(\mathit{Abloy}, 4, \mathit{Short}, \mathit{Broad}), (\mathit{Chubb}, 3, \mathit{Medium}, \mathit{Broad}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Rubo}, 5, \mathit{Long}, \mathit{Narrow})\} = \bot.
\end{aligned}
$$

The problem can now be stated more precisely. It is to find a definition for the function $\mathit{opens} : \mathit{Bunch} \rightarrow \Omega$ that is consistent with the above examples and correctly predicts whether or not new bunches of keys will open the door.

Stated this way, the problem is one of induction: given values of the function on some specific individuals, find the general definition of the function. In practical applications, there are usually a very large number of definitions that are consistent with the examples, so, to make any progress, it is necessary to make further assumptions. These assumptions constrain the possible definitions of the function, that is, the so-called *hypotheses*, that will be admitted. The form of the possible hypotheses is specified by the *hypothesis language*. It is a general principle of learning that, in order to learn at all, one must make some assumptions about the hypothesis language.

So let's consider what might be a suitable hypothesis language for the illustration under investigation. If the door is a standard one with a single

keyhole, it seems reasonable to assume that a bunch of keys opens the door if and only if (iff) there is some key on the bunch that opens the door. So the problem reduces to finding a key on the bunch with some property. Now what properties might an individual have? Given that a key is represented by four characteristics, the assumption is made that a suitable property is a conjunction of conditions on these characteristics. Finally, a condition on a characteristic is assumed to be of the form whether that characteristic is equal to some constant. To precisely specify the hypothesis language, it is necessary to formally state these kinds of restrictions.

To get a condition on a key, one must have access to the components of the key. This suggests introducing the projections from the keys onto each of their components.

$$
\begin{aligned}
&projMake : Key \rightarrow Make \\
&projNumProngs : Key \rightarrow NumProngs \\
&projLength : Key \rightarrow Length \\
&projWidth : Key \rightarrow Width.
\end{aligned}
$$

For example, *projMake* is the projection from keys onto their first component.

With these projections available, it is easy to impose a condition on a key. For example, the condition that the make of a key k should be Abloy is expressed by

$$(projMake\ k) = Abloy.$$

Conjunctions of such conditions are also admitted into the hypothesis language by including the conjunction connective $\wedge$. Then, according to the assumptions made on the hypothesis language given above, conditions on bunches have the form

$$\exists k.((k \in b) \wedge C),$$

where $\exists$ is the existential quantifier, so that '$\exists k$.' means 'there is a k such that', '$k \in b$' means 'k is a member of the set b', '$\wedge$' means 'and', and C is some condition of the form above on keys.

Given the set of examples and the form that the hypothesis language can take, as stated above, a learning system can now try to induce a suitable definition for *opens*. The ALKEMY learning system studied later in this book finds the following hypothesis (although not in exactly this form; a more convenient syntax will be introduced later).

$$
\begin{aligned}
&opens\ b = \\
&\quad if\ \exists k.((k \in b) \wedge ((projMake\ k) = Abloy) \wedge ((projLength\ k) = Medium)) \\
&\quad then\ \top \\
&\quad else\ \bot.
\end{aligned}
$$

This definition can be translated straightforwardly into (structured) English that can be understood by someone who is not familiar with the knowledge representation language employed, as follows.

> "A bunch of keys opens the door if and only if it contains an Abloy key of medium length".

One can see that this definition agrees with all the examples given earlier and thus it is potentially a suitable definition for *opens*. Whether or not it correctly predicts that new bunches will open the door can only be checked by trying the definition on these examples, the so-called *test data*.

Learning Issues

This simple illustration has highlighted most of the important issues to be studied in this book.

First, the *individuals* that are the subject of learning have to be represented. In the illustration above, the individuals were represented by sets of tuples of constants. In general, many other types are also needed such as lists, trees, multisets, and graphs. The formalism of Chap. 3 provides a suitable knowledge representation language for representing individuals and is particularly concerned with the case where the individuals have complex structure.

Second, one has to specify a *signature* for the *target function* whose definition is to be induced. This signature states that the function maps from the type of the individuals to the type of a (usually small) finite set that consists of the so-called *classes* to which the individuals can be mapped. Often there are only two classes and one uses $\bot$ and $\top$ (or 0 and 1, or -1 and 1) to denote them. Such learning problems are called *classification problems*. In a *regression problem*, the codomain of the target function is the real numbers.

Third, there is given some *training data*, which is a collection of *examples* each of which gives the value of the target function for a particular individual. For the illustration above, there are only nine examples; in practical applications, there may be hundreds or thousands of examples available for training. (In the case of data mining, there may be millions of training examples.) In general, the more numerous the training data the better. Learning with such training data is called *supervised learning*. In some problems only the individuals are given without a value of some function. In this case, the problem is one of *unsupervised learning* and one usually wants to somehow cluster the individuals appropriately.

Fourth, the so-called *background theory* must be given. This theory consists of the definitions of functions that act on the individuals (together with associated functions). For example, for the illustration above, the function *projMake* is in the background theory and its definition is simply

$$projMake\ (x_1, x_2, x_3, x_4) = x_1.$$

For some applications, the background theory may be very extensive. Generally it can be divided into two parts. The *generic background theory* is the part of the background theory that is dependent only on the type of the individuals. For example, if the individuals are graphs, then the background theory could contain the function that maps a graph to its set of vertices. If the individuals are lists, then the background theory could contain the function that maps a (non-empty) list to its head. In contrast, the *domain-specific background theory* is the part of the background theory that depends on knowledge associated with the particular application. For example, if the application involves the carcinogenicity of chemical molecules, then the domain-specific background theory could contain the function that maps a molecule to the number of benzene rings that it contains. An important point that will be developed in this book is that much of the background theory is largely *determined* by the type of the individuals. However, the key to learning is often in the domain-specific part for which expert knowledge may be required.

Fifth, the *hypothesis language* that involves the functions in the background theory must be specified. In Chap. 4, a mechanism will be introduced for precisely stating hypothesis languages and also for enumerating hypotheses in the language. *Bias* restricts the form of potential hypotheses and comes in one of two forms: *language bias* that determines the hypothesis language itself and *search bias* that determines the way the learning system searches the hypothesis space for a suitable hypothesis. In practical applications, this space can be huge and, therefore, it may be necessary to search it preferentially, prune subspaces of it based on certain criteria, and so on.

Sixth, one has to evaluate the predictive power of the hypothesis constructed. Here, one is concerned with how well an hypothesis *generalises*, that is, correctly predicts the class of new individuals. Typically, hypotheses are evaluated experimentally by systematically trying them on test data, by cross-validation techniques, for example. Also, it may be important to determine experimentally how the predictive power improves as the size of the training data increases. Finally, it may be possible to estimate analytically the predictive power of an hypothesis by studying some characteristics of the hypothesis language.

Seventh, it is often desirable and sometimes essential that the hypothesis returned by the learning system be *comprehensible*, that is, be easily understandable by humans in such a way that it provides insight into, or an 'explanation' of, the data. Whether comprehensibility is really required depends on the application: sometimes one is satisfied with a black-box that has good predictive power, even if the reasons for the good predictive power are unclear; sometimes, especially in applications to expert systems, scientific discovery, and intelligent agents, comprehensibility is essential. This book concentrates on the case when comprehensibility is required, but also considers some learning methods that do not have this characteristic.

There are many different kinds of learning systems employed in applications. These include neural networks, decision-tree systems, instance-based learners based on a distance measure between individuals, kernel-based learners based on a generalised inner product defined on individuals, and learners based on Bayesian principles. Furthermore, learning systems can be classification systems that learn a function that maps into some set of classes, regression systems that learn a function that maps into the real numbers, or systems that cluster individuals that are not labelled in any way.

1.4 Introduction to Logic

Examining the uses of knowledge representation in Sect. 1.3, several requirements become apparent: the formalism must be able to represent complex individuals, there should be a way of precisely stating the hypothesis language, and it must be possible to compute the value of a function in the background theory on an individual. While there are other possible approaches, this book takes the view that higher-order logic conveniently meets all the above requirements. Consequently, in this section, I outline the basic features of higher-order logic that are needed for learning applications.

Terms and Types

Logics, in general, have two fundamental aspects: syntax and semantics. The syntax is concerned with what expressions are defined to be well-formed, what formulas (that is, terms having boolean type) are theorems, and proofs of those theorems. The semantics is concerned with the meanings of the symbols in the terms and the terms themselves, what are the interpretations that give those meaning, what are the models (that is, the interpretations that make all the axioms true), and what formulas are valid (that is, true in every possible model). In this introduction, I concentrate on syntax starting with the concept of a term.

First, some symbols must be made available. Thus it is assumed that there is given an alphabet of symbols that include some variables and some constants (amongst some other symbols that will be introduced later). Then the terms can be (informally) defined as follows. A variable is a term; a constant is a term; an expression of the form $\lambda x.t$ is a term, where x is a variable and t is a term; an expression of the form $(s\ t)$ is a term, where s and t are terms; and an expression of the form $(t_1, \ldots, t_n)$ is a term, where $t_1, \ldots, t_n$ are terms.

Variables and constants play the part one would expect in the logic. (There are plenty of constants in Sect. 1.3 and use was made there of variables, as well.) Terms of the form $\lambda x.t$ are called *abstractions* and come originally from the λ-calculus of Church. The meaning of $\lambda x.t$ is that it is a function

that maps an element denoted by a term s to the element denoted by the term obtained by replacing each free occurrence of x in t by s. So, for example, the meaning of $\lambda x.(x + 1)$ is the function that increments the value of its argument. (Note that an occurrence of a variable y is bound if it occurs with a subterm of the form $\lambda y.s$; otherwise, the occurrence is free. A variable is free if it has a free occurrence.) A term of the form $(s\ t)$ is an *application* in which s is applied to t. Thus the meaning of s should be a function, the meaning of t should be an argument to the function, and the meaning of $(s\ t)$ should the the result of applying s to t. For example, the meaning of $(\lambda x.(x + 1)\ 42)$ is 43. Finally, a term of the form $(t_1, \ldots, t_n)$ is a *tuple.*

What has been described so far is essentially the terms of the *untyped* λ-calculus. However, in knowledge representation applications in computer science, it is important to impose some further restrictions on the terms that are admitted. The reason is that, with the definition so far, some strange terms are allowed. For example, there is no restriction that the argument to a function necessarily belong to the domain of that function, whatever that might be. To give an example, what might $(\lambda x.(x + 1)\ \mathit{Abloy})$ mean? Thus, types are introduced to restrict term formation to terms that make intuitive sense from this point of view. As shall be seen, the discipline of types pervades the book and there will be a substantial payoff in accepting this discipline. I now introduce types.

For this purpose, suppose the alphabet is enlarged with some extra symbols called *type constructors*, each of which has an arity that determines the number of arguments to which the type constructor can be applied. Typical type constructors of arity 0 include Ω, the type of the booleans, *Int*, the type of the integers, and *Char*, the type of the characters. A typical type constructor of non-zero arity is *List* of arity 1.

One can then define types as follows: $T\ \alpha_1 \ldots \alpha_n$ is a type, where T is a type constructor of arity n and $\alpha_1, \ldots, \alpha_n$ are types; $\alpha \to \beta$ is a type, where α and β are types; and $\alpha_1 \times \cdots \times \alpha_n$ is a type, where $\alpha_1, \ldots, \alpha_n$ are types.

The first part of the definition implies that nullary type constructors are types. Thus Ω and *Int* are types. Since *List* is a unary type constructor, it follows that *List* Ω and *List Int* are types. The meaning of a type is a set. In particular, the meaning of *Int* is the set of integers and the meaning of *List Int* is the set of lists of integers. The meaning of a type of the form $\alpha \to \beta$ is a set of functions from the set giving the meaning of α to the set giving the meaning of β. The meaning of a type of the form $\alpha_1 \times \cdots \times \alpha_n$ is the cartesian product of the sets that give the meanings of $\alpha_1, \ldots, \alpha_n$.

With types now in place, I revisit the definition of terms. To impose a type discipline, one starts by giving types to the variables and constants. For the variables, one assumes that for each type, there is associated a disjoint set of variables for that type. For the constants, one assumes that for each constant, there is specified some type, called its *signature*. For example, from Sect. 1.3, *Abloy* is a constant with signature *Make* and *opens* is a constant

with signature *Bunch* $\rightarrow \Omega$. That a constant C has signature α is denoted by $C : \alpha$.

Now the definition of terms that respect the typing discipline can be given. A variable of type α is a term of type α; a constant with signature α is a term of type α; an expression of the form $\lambda x.t$, where x is a variable of type α and t is a term of type β, is a term of type $\alpha \rightarrow \beta$; an expression of the form $(s\ t)$, where s is a term of type $\alpha \rightarrow \beta$ and t is a term of type α, is a term of type β; and an expression of the form $(t_1, \ldots, t_n)$, where t_i is a term of type α_i, for $1 = 1, \ldots, n$, is a term of type $\alpha_1 \times \cdots \times \alpha_n$.

Example 1.4.1. Suppose the alphabet contains the type constructors Ω, *Int*, and *List*, and the constants

$$
\begin{aligned}
&[] : \mathit{List}\ \mathit{Int}, \\
&\sharp : \mathit{Int} \rightarrow (\mathit{List}\ \mathit{Int} \rightarrow \mathit{List}\ \mathit{Int}), \\
&p : \mathit{List}\ \mathit{Int} \rightarrow \Omega, \\
&q : \mathit{List}\ \mathit{Int} \rightarrow \Omega, \text{ and} \\
&\wedge : \Omega \rightarrow (\Omega \rightarrow \Omega).
\end{aligned}
$$

The intended meaning of [] is the empty list and $\sharp$ is the constant used for constructing lists. Thus $((\sharp\ 1)\ ((\sharp\ 2)\ ((\sharp\ 3)\ [])))$ is intended to represent the list $[1, 2, 3]$. (Since this notation is rather heavy, I usually drop the parentheses and write $\sharp$ as an infix operator. Thus the expression above can be written more simply as $1 \sharp 2 \sharp 3 \sharp []$.)

Some remarks about the ubiquitous use of $\rightarrow$ in type declarations are in order. It may seem more natural to use the signature

$$\mathit{Int} \times \mathit{List}\ \mathit{Int} \rightarrow \mathit{List}\ \mathit{Int}$$

for $\sharp$. With this signature, $\sharp$ is intended to take an item and a list as input and return the list obtained by prepending the given item to the given list. However, the signature *Int* $\rightarrow$ (*List Int* $\rightarrow$ *List Int*) for $\sharp$, the so-called *curried* form of the signature, is actually more convenient. The reason is that, in the curried form, one only has to give $\sharp$ one argument at a time. Thus $(\sharp\ 3)$ is well-defined and, in fact, is a term of type *List Int* $\rightarrow$ *List Int*. Thus one can then form $((\sharp\ 3)\ [])$ which is a term of type *List Int*. Similarly, $(\sharp\ 2)$ is well-defined and $((\sharp\ 2)\ ((\sharp\ 3)\ []))$ is a term of type *List Int*, and so on. Throughout the book, wherever possible, curried signatures will be used.

Assume now that x is a variable of type *Int*, and y and z are variables of type *List Int*. I show that the expression

$$((\wedge\ (p\ ((\sharp\ x)\ y)))\ (q\ z))$$

is a term of type Ω. (Exploiting the infix use of $\sharp$ and $\wedge$, this expression can be written more simply as $(p\ (x \sharp y)) \wedge (q\ z)$.) First, $(\sharp\ x)$ is a term of type

List Int $\rightarrow$ *List Int*, $((\sharp\ x)\ y)$ is a term of type *List Int*, and so $(p\ ((\sharp\ x)\ y))$ is a term of type Ω. Also, $(q\ z)$ is a term of type Ω. Now $(\wedge\ (p\ ((\sharp\ x)\ y)))$ is a term of type $\Omega \rightarrow \Omega$, and thus $((\wedge\ (p\ ((\sharp\ x)\ y)))\ (q\ z))$ is a term of type Ω.

The logic is suitable for writing definitions in functional programming languages.

Example 1.4.2. Consider the function

$$concat : List\ Int \rightarrow (List\ Int \rightarrow List\ Int),$$

whose intended meaning is that the result of applying the function to two lists (of integers) is their concatenation. A suitable definition for this function is

$$concat\ []\ z = z$$
$$concat\ (x \sharp y)\ z = x \sharp (concat\ y\ z).$$

(Here I have exploited the convention that function application is left associative so that *concat* $[]\ z$ means $((concat\ [])\ z)$, and so on.) Each equation in the definition is true using the intended meaning of *concat*.

So far it is not obvious how the quantifiers are introduced into the logic. Existential quantification is considered first. Let Σ be the function having signature

$$(Int \rightarrow \Omega) \rightarrow \Omega$$

and with the intended meaning that Σ maps a predicate of type $Int \rightarrow \Omega$ to true iff the predicate is true on at least one element in its domain. (A predicate is a function whose codomain is the booleans.) Suppose now that $r : Int \rightarrow \Omega$ is a predicate and consider the term $(\Sigma\ \lambda x.(r\ x))$. According to the meaning of Σ, this term will be true iff there is an x for which $(r\ x)$ is true. In other words, the intended meaning of $(\Sigma\ \lambda x.(r\ x))$ is exactly the same as $\exists x.(r\ x)$, using the standard meaning of existential quantification.

A term of the form $(\Sigma\ \lambda x.t)$ is written as $\exists x.t$. Thus the existential quantifier is introduced into higher-order logic by the function Σ applied to an abstraction whose body is a formula. The binding aspect of the quantifier is taken care of by the λ-expression and the precise form of quantification by the Σ.

Similarly, let Π be the function having signature

$$(Int \rightarrow \Omega) \rightarrow \Omega$$

and with the intended meaning that Π maps a predicate to true iff the predicate is true on all elements in its domain. According to the meaning of Π, $(\Pi\ \lambda x.(r\ x))$ will be true iff $(r\ x)$ is true, for every x. In other words, the intended meaning of $(\Pi\ \lambda x.(r\ x))$ is exactly the same as $\forall x.(r\ x)$.

A term of the form $(\Pi\ \lambda x.t)$ is written as $\forall x.t$. Thus the universal quantifier is introduced by the function Π applied to an abstraction whose body is a formula. In an analogous way to existential quantification, the binding aspect of the quantifier is taken care of by the λ-expression and the precise form of quantification by the Π.

Polymorphism

The logic introduced so far is expressive and powerful: all the usual connectives and quantifiers are available and (many-sorted) first-order logic is a subset. It *would* be possible to use this logic as the setting for the remainder of this book and many of the following technical results (in Chap. 2, for example) would actually become much simpler. But the logic has one restriction that makes this approach unattractive. To make the point, consider the function *concat* defined above that concatenates lists of integers. A moment's thought reveals that the definition given for *concat* works perfectly well for lists whose items are of *any* type. Thus, with the current version of the logic, one would be forced to declare a *concat* function for each of the types one wanted to apply it to and yet the definitions of the various *concat*s would all be exactly the same. One could make a similar comment about the functions Σ and Π that work perfectly well when quantifying over variables of any type.

For this reason, one final feature of the logic is now introduced: *polymorphism*, or more precisely because there are other forms of polymorphism, *parametric polymorphism.* Thus the last ingredient of the alphabet are *parameters*, which are type variables. Parameters are usually denoted by a, b, and so on. For example, one can make *concat* polymorphic by declaring it to have the signature

$$List\ a \to (List\ a \to List\ a).$$

The intended meaning of the signature is that it declares a *concat* function for each possible instantiation of the parameter a. Similarly, the signature of the polymorphic version of both Σ and Π is

$$(a \to \Omega) \to \Omega.$$

The introduction of polymorphism requires extensions of some previous definitions. For a start, the definition of types is extended as follows. A parameter is a type; $T\ \alpha_1 \ldots \alpha_n$ is a type, where T is a type constructor of arity n and $\alpha_1, \ldots, \alpha_n$ are types; $\alpha \to \beta$ is a type, where α and β are types; and $\alpha_1 \times \cdots \times \alpha_n$ is a type, where $\alpha_1, \ldots, \alpha_n$ are types.

But the biggest complication occurs in the definition of terms. To illustrate the point, with the functions $r : Int \to \Omega$ and $\Sigma : (a \to \Omega) \to \Omega$ available, consider whether the expression $(\Sigma\ \lambda x.(r\ x))$ should be a term or not. Now

$\lambda x.(r\ x)$ is a term of type $Int \to \Omega$, so that one is led to compare this type with the argument type $a \to \Omega$ of Σ. The question is whether these two types are somehow 'compatible'. The intuitive notion of compatibility here is that there should be an instantiation of the parameters in the two types so that the argument type (that is, $Int \to \Omega$) is the same as the domain type of Σ (that is, $(a \to \Omega)$). The substitution $\{a/Int\}$, in which a is replaced by Int, shows that the types are indeed compatible. Thus $(\Sigma\ \lambda x.(r\ x))$ should be a term of type Ω, for which the specific Σ function being used here is the one with signature $(Int \to \Omega) \to \Omega$.

More generally, one could have terms s of type $\alpha \to \beta$ and t of type γ (where s and t do not share free variables) and ask whether $(s\ t)$ should be a term. The answer to the question will be yes iff the types α and γ have a common instance. In case there is a common instance, then α and γ have a most general unifier θ, say. (A most general unifier is a substitution that when applied makes the two instances identical and is a substitution that instantiates as little as possible in order to achieve this.) Then $(s\ t)$ is a term of type $\beta\theta$.

Example 1.4.3. Let s be a term of type $(List\ a \times Int) \to List\ a$ and t a term of type $List\ \Omega \times b$ (that do not share free variables). Now $\{a/\Omega, b/Int\}$ is a most general unifier of $List\ a \times Int$ and $List\ \Omega \times b$. Thus $(s\ t)$ is a term of type $(List\ a)\{a/\Omega, b/Int\} = List\ \Omega$.

There is one other form of compatibility that has to be dealt with when forming terms and that concerns free variables.

Example 1.4.4. Let $p : Int \to \Omega$ and $q : List\ Int \to \Omega$ be predicates. Is it reasonable that the expression $(p\ x) \wedge (q\ x)$ be a term? Even without a precise definition of the notion of a term, one would expect this expression to be problematic. To see this, note the two free occurrences of the variable x. Now a principle of the formation of terms in logics is that all free occurrences of a variable should denote the same individual. But this does not hold in $(p\ x) \wedge (q\ x)$ because the first occurrence of x has type Int because it is the argument of p, while the second has type $List\ Int$ because it is the argument of q. Thus the expression should not be a term.

Now an informal definition of a term for the polymorphic version of the logic can be given. For this, there is a single family of variables and the constants may have polymorphic signatures. Free variables in a term have a relative type according to their position in the term. For example, if $p : Int \to \Omega$, then the free variable x in $(p\ x)$ has relative type Int in $(p\ x)$. Then the definition is as follows. A variable is a term of type a, where a is a parameter; a constant with signature α is a term of type α; an expression of the form $\lambda x.t$, where x is free with relative type type α in t and t is a term of type β, is a term of type $\alpha \to \beta$; an expression of the form $(s\ t)$, where s is a term of type $\alpha \to \beta$ and t is a term of type γ, is a term of type $\beta\theta$,

provided there is a most general unifier θ of the set of equations that arise from unifying α and γ, and also unifying the relative types of occurrences of free variables in s and t ; and an expression of the form $(t_1, \ldots, t_n)$, where t_i is a term of type α_i, for $1 = 1, \ldots, n$, is a term of type $(\alpha_1 \times \cdots \times \alpha_n)\theta$, provided there is a most general unifier θ of the set of equations that arise from unifying relative types of occurrences of free variables in the $t_1, \ldots, t_n$.

Example 1.4.5. Let $p : List\ a \rightarrow \Omega$ and $q : List\ Int \rightarrow \Omega$ be predicates. Then $(p\ x)$ is a term of type Ω in which the free variable x has relative type $List\ a$, $(q\ x)$ is a term of type Ω in which the free variable x has relative type $List\ Int$, $\lambda x.(p\ x)$ is a term of type $List\ a \rightarrow \Omega$, $\exists x.(p\ x)$ is a term of type Ω, $((p\ x), (q\ x))$ is a term of type $\Omega \times \Omega$ in which the free variable x has relative type $List\ Int$, and $(p\ x) \wedge (q\ x)$ is a term of type Ω in which the free variable x has relative type $List\ Int$.

Logic as a Computational Formalism

The logic is also suitable as a formalism in which to write definitions of functions for declarative programming languages.

Example 1.4.6. Consider again the function

$$concat : List\ a \rightarrow (List\ a \rightarrow List\ a)$$

defined by

$$concat\ []\ z = z$$
$$concat\ (x \sharp y)\ z = x \sharp (concat\ y\ z).$$

This definition can be used by a declarative programming language to concatenate lists. For example, one can concatenate the lists $1 \sharp 2 \sharp []$ and $3 \sharp []$ by the computation

$$concat\ (1 \sharp 2 \sharp [])\ (3 \sharp [])$$
$$1 \sharp (concat\ (2 \sharp [])\ (3 \sharp []))$$
$$1 \sharp 2 \sharp (concat\ []\ (3 \sharp []))$$
$$1 \sharp 2 \sharp 3 \sharp [],$$

whereby the initial term is successively ‘simplified’ by rewriting steps using the equations in the definition of *concat*.

Example 1.4.7. The function

$$length : List\ a \rightarrow Int$$

$$length\ [] = 0$$
$$length\ (x \sharp y) = 1 + length\ y$$

computes the length of a list.

The following example is a more complicated one that illustrates the use of existential quantification.

Example 1.4.8. The function

$$append : List\ a \times List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} append\ (u, v, w) &= (u = [] \wedge v = w) \vee \\ &\exists r.\exists x.\exists y.(u = r \sharp x \wedge w = r \sharp y \wedge append\ (x, v, y)) \end{aligned}$$

returns true iff its third argument is the concatenation of the lists in its first and second arguments. As an example of a computation, the term

$$append\ (1 \sharp 2 \sharp [], 3 \sharp [], x)$$

can be simplified to

$$x = 1 \sharp 2 \sharp 3 \sharp [],$$

using this definition. Similarly, the term

$$append\ (x, y, 1 \sharp 2 \sharp [])$$

can be simplified to

$$(x = [] \wedge y = 1 \sharp 2\ []) \vee (x = 1 \sharp [] \wedge y = 2 \sharp []) \vee (x = 1 \sharp 2 \sharp [] \wedge y = []).$$

Representation of Individuals

The next topic is that of the representation of individuals. Learning (and other) applications involve individuals of many different kinds. Thus the challenge is to find a suitable class of (higher-order) terms to represent this wide range of individuals.

The formal basis for the representation of individuals is provided by the concept of a *basic term.* Having defined the concept of a term, basic terms are defined via an inductive definition that has three parts: the first part gives those basic terms that have a data constructor (explained below) at the top level, the second part gives certain abstractions that include (finite) sets and multisets, and the third part gives tuples. Care is taken to order certain subterms of abstractions to ensure uniqueness of the representation.

The simplest kinds of individuals can be represented by terms of 'atomic' types such as integers, natural numbers, floating-point numbers, characters, strings, and booleans. Closely related are lists and trees. The constants

$[] : List\ a,$ and

$\sharp : a \rightarrow List\ a \rightarrow List\ a$

are used to represent lists. Similarly, one could use the unary type constructor *Tree* and the constants

$$Null : Tree\ a, \text{ and}$$
$$Node : Tree\ a \rightarrow a \rightarrow Tree\ a \rightarrow Tree\ a$$

to represent binary trees in the obvious way.

The constants of the logic are divided into two kinds: functions and data constructors. Functions have definitions and are used to compute values. Some earlier examples of functions are *concat* and *length*. In contrast, data constructors are used, as their name implies, to construct data. Typical examples come from the previous paragraph: numbers, characters, strings, [], $\sharp$, *Null*, and *Node*. In general, data constructors have signatures of the form

$$\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k),$$

for some k-ary type constructor T, types $\sigma_1, \ldots, \sigma_n$, and parameters $a_1, \ldots, a_k$. A data constructor with such a signature is said to have *arity* n. The first part of the definition of basic terms states roughly that, if C is a data constructor of arity n and $t_1, \ldots, t_n$ are basic terms having suitable types, then $C\ t_1 \ldots t_n$ is a basic term.

Example 1.4.9. With the declarations of *Null* and *Node* above,

$$Node\ (Node\ Null\ 21\ Null)\ 42\ (Node\ Null\ 73\ Null)$$

is the basic term representing the tree with 42 at the root, 21 at the left child of the root, and 73 at the right child.

The second kind of basic term are sets, multisets, and similar types. To explain how these types are handled, I concentrate on sets. The first question is what exactly is a set? The answer to this question for a higher-order logic is that a set is a predicate, that is, a set is identified with its characteristic function. Thus particular forms of abstractions are used to represent sets. To explain this, consider the set $\{1, 2\}$. This is represented by the abstraction

$$\lambda x.if\ x = 1\ then\ \top\ else\ if\ x = 2\ then\ \top\ else\ \bot.$$

The meaning of this abstraction is the predicate that is true on 1 and 2, and is false for all other numbers. Similarly,

$$\lambda x.if\ x = A\ then\ 42\ else\ if\ x = B\ then\ 21\ else\ 0$$

is the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus abstractions of the form

$$\lambda x.if\ x = t_1\ then\ s_1\ else\ \ldots\ if\ x = t_n\ then\ s_n\ else\ s_0$$

are adopted to represent (finite) sets, multisets, and so on. The term s_0 is a *default term*, $\bot$ for sets and 0 for multisets. Other types have a different default term.

This discussion leads to the second part of the definition of basic terms. Roughly speaking, if $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ are basic terms having suitable types and s_0 is a default term, then

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

is a basic term. The precise definition takes care to put an order on $t_1, \ldots, t_n$, amongst some other things.

Finally, tuples of basic terms are basic terms. Thus, if $t_1, \ldots, t_n$ are basic terms, then $(t_1, \ldots, t_n)$ is a basic term.

Example 1.4.10. The expression

$$\{(\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}), (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}),\\ (\mathit{Chubb}, 3, \mathit{Long}, \mathit{Narrow})\}$$

is notational sugar for the basic term

$$\begin{aligned}\lambda x.\mathit{if}\ x = (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal})\ \mathit{then}\ \top\ \mathit{else}&\\ \mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}&\\ \mathit{if}\ x = (\mathit{Chubb}, 3, \mathit{Long}, \mathit{Narrow})\ \mathit{then}\ \top\ \mathit{else}\ \bot.&\end{aligned}$$

Other notational devices are noted here. Having identified a set with a predicate, the type of a set has the form $\alpha \to \Omega$, for some type α. However, it is still useful to think of sets as predicates in some circumstances and as 'collections of elements' in other circumstances. In this second circumstance, it is convenient to introduce the notational sugar $\{\alpha\}$ to mean $\alpha \to \Omega$. Also, if a set t is being thought of as a predicate, then application is denoted in the usual way by $(t\ x)$, while if t is being thought of as a collection of elements, this same term is denoted by $x \in t$. Advantage will be taken of these notational devices shortly.

Predicate Construction

The final topic of this section is a brief explanation of how the higher-order nature of the logic can be exploited to construct predicates. To make the ideas concrete, the keys illustration in Sect. 1.3 is revisited. The condition that appears there in the induced definition for *opens* is

$$\exists k.((k \in b) \wedge ((\mathit{projMake}\ k) = \mathit{Abloy}) \wedge ((\mathit{projLength}\ k) = \mathit{Medium})).$$

A more convenient reformulation of this condition is now explored.

For each constant of type *Make*, there is a corresponding predicate that is true iff its argument is equal to the constant. For example, corresponding to the constant *Abloy*, there is a predicate

$$(= Abloy) : Make \rightarrow \Omega$$

defined by

$$((= Abloy)\ x) = x = Abloy.$$

Thus $((= Abloy)\ x) = \top$ iff $x = Abloy$. More generally, given a constant $C : \alpha$, there corresponds a predicate $(= C) : \alpha \rightarrow \Omega$ defined by $((= C)\ x) = x = C$.

Conditions on the characteristics of a key can be obtained by composing a projection with one of the predicates just introduced. Composition is given by the (reverse) composition function

$$\circ : (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c)$$

defined by

$$((f \circ g)\ x) = (g\ (f\ x)).$$

Note the order here: for $f \circ g$, f is applied first, then g. Thus one can form a predicate such as $projMake \circ (= Abloy)$ that has type $Key \rightarrow \Omega$. If k is a key, then $((projMake \circ (= Abloy))\ k) = \top$ iff the first component of k is *Abloy*.

Next consider the connective $\wedge : \Omega \rightarrow (\Omega \rightarrow \Omega)$ in the condition in the definition for *opens*. Connectives act on formulas; what is needed is to 'lift' the connectives to functions that act on predicates. Thus consider the function

$$\wedge_2 : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow a \rightarrow \Omega$$

defined by

$$\wedge_2\ p\ q\ x = (p\ x) \wedge (q\ x).$$

Using $\wedge_2$, one can form the predicate

$$\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium)).$$

This predicate is true on a key iff the first component of the key is *Abloy* and the third component is *Medium*.

The final step in the reformulation of the condition in the definition of *opens* is to replace the $\exists k.(k \in b)$ part of it. Consider the function

$$setExists_1 : (a \rightarrow \Omega) \rightarrow \{a\} \rightarrow \Omega$$

defined by

$$setExists_1\ p\ t = \exists x.((p\ x) \wedge (x \in t)).$$

(Note the use of the notation $\{a\}$ here; the first argument to $setExists_1$ is being thought of as a predicate and the second is being thought of as a collection of elements.) The predicate $(setExists_1\ p)$ checks whether a set has an element that satisfies p. Thus we can form the predicate

$$setExists_1\ (\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium))).$$

Note now that the condition

$$(setExists_1\ (\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium)))\ b)$$

is equivalent to

$$\exists k.((k \in b) \wedge ((projMake\ k) = Abloy) \wedge ((projLength\ k) = Medium)).$$

This completes the reformulation of the condition in the definition of *opens*.

What has been achieved by reformulating the condition in this way? The major gain is that it provides the basis for a convenient way of constructing predicates. In this approach, predicates are constructed from other predicates by composition. Thus one starts with some 'atomic' predicates and forms more complex predicates by systematically composing them. A key definition to make all this work is the following. A *transformation* f is a function having a signature of the form

$$f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \mu \to \sigma,$$

where $k \geq 0$. Clearly, $\wedge_2$ and $setExists_1$ are transformations and many more are introduced later in the book. In general, if $p_i : \varrho_i \to \Omega\ (i = 1, \ldots, n)$, then $f\ p_1 \ldots p_n : \mu \to \sigma$, and several such functions, the last of which is a predicate, can be composed to form a predicate. A method is also introduced whereby the hypothesis language is specified by a system of rewrites and predicates in the hypothesis language are systematically constructed by a rewriting process that exploits composition. This approach allows precise and explicit control over the hypothesis language.

Bibliographical Notes

For an account of the history of computational logic, see [77]. A standard and comprehensive reference on artificial intelligence that gives a brief history of artificial intelligence, including machine learning and the various logical influences, is [79].

Excellent general accounts of machine learning are in [61] and [79]. To properly understand machine learning, it is essential to have a good understanding of its theory, computational learning theory. With some reluctance,

I decided not to include a discussion of this theory, partly because it seemed better not to distract from the book's concentration on knowledge representation issues and partly because other authors have already written much better accounts than I ever could. For readers not familiar with this material, I suggest starting with the introductory account in Chap. 7 in [61]. A more sophisticated account with many references is in Chap. 5 in [84]. As a guide to what to read about, here is a list of key concepts in computational learning theory: risk, empirical risk, overfitting, sample complexity, consistency, empirical risk minimisation, structural risk minimisation, PAC-learnable, capacity, VC dimension, and model selection.

Plotkin's work on induction is described in [71] and [72], while Reynold's is in [76]. Some of Michalski's early work on induction is summarised in [58]. The work of Vere on inducing concepts in first-order logic is in [88]. The Marvin system is discussed in [80] and [82]. The model inference system of Shapiro is described in [85]. Buntine's work on generalised subsumption is in [10]. The contribution of inverse resolution by Muggleton and Buntine is described in [64]. The Foil system of Quinlan appears in [74].

The field of inductive logic programming was christened in [63]. The first comprehensive account of its research agenda was given in [65]. The theoretical foundations of logic programming are described in [52] and those of inductive logic programming are in [70]. An excellent recent account of the inductive logic programming perspective on data mining and much more besides is in [20]. The home page of the European Network of Excellence in Inductive Logic Programming is [38]. Details of the ILP workshop series are recorded at [37]. Histories of inductive logic programming can be found in [63], [70], and [81].

Type theory arose from attempts to provide a foundation for mathematics at the beginning of the 20th Century. Russell's paradox and the more general crisis in set theory at the time led to the introduction of various type disciplines to circumvent the problems [78]. The original account of the simple theory of types is in [11] and its model theory is given in [34]. Modern accounts of higher-order logic are in [1] and [90], while categorical treatments can be found in [2] and [46].

A survey of functional logic programming up to 1994 is in [31]. Haskell is described in [39], λProlog in [68], HOL in [30], and Curry in [32].

Exercises

1.1 Read the brief history of artificial intelligence in [79, Chap. 1]. Comment on the changing role of logic in artificial intelligence and machine learning over the last 40 years.

1.2 (For those who know machine learning.) Enumerate the techniques you know about for representing structured data in learning applications. Typical

applications involve chemical molecules or DNA strings in bioinformatics and XML or HTML pages for the World Wide Web. When you have finished reading this book, discuss the pros and cons of the techniques you have listed compared with the ones introduced here.

1.3 (For those who know computational logic.) Enumerate the advantages and disadvantages of first-order compared with higher-order logic for computational logic applications such as declarative programming languages or theorem proving systems. When you have finished reading this book, investigate whether your earlier analysis needs modification.

1.4 For the keys example of Sect. 1.3, give two other possible hypothesis languages. Can you give an hypothesis language for which one can express the definition that *opens* is $\top$ on a bunch iff it is one of the training examples that opens the door? If so, what implications might this have for learning a definition that generalises to so far unseen individuals?

1.5 Consider the definition of the *append* function given in Sect. 1.4. Investigate how the term $append\ (1 \sharp 2 \sharp [], 3 \sharp [], x)$ might be simplified to $x = 1 \sharp 2 \sharp 3 \sharp []$.
[Hint: You will need to invent some suitable equations for the definitions of $\exists$ and $=$. For example, what (general) equation for $\exists$ would allow, say,

$$\exists x.\exists y.(x = 1 \sharp [] \ \wedge \ append\ ([], x, y))$$

to be reduced in one step to

$$\exists y.append\ ([], 1 \sharp [], y)?]$$

2. Logic

This chapter contains an account of the aspects of the syntax and semantics of higher-order logic that are most relevant to its application to learning.

2.1 Types

Definition 2.1.1. An *alphabet* consists of four sets:

1. A set $\mathfrak{T}$ of type constructors.
2. A set $\mathfrak{P}$ of parameters.
3. A set $\mathfrak{C}$ of constants.
4. A set $\mathfrak{V}$ of variables.

Each type constructor in $\mathfrak{T}$ has an arity. The set $\mathfrak{T}$ always includes the type constructors *1* and Ω both of arity 0. A *nullary* type constructor is a type constructor of arity 0. *1* is the type of some distinguished singleton set and Ω is the type of the booleans. The set $\mathfrak{P}$ is denumerable (that is, countably infinite). Parameters are type variables and are typically denoted by $a, b, c, \ldots$. Each constant in $\mathfrak{C}$ has a signature (see below). The set $\mathfrak{V}$ is also denumerable. Variables are typically denoted by $x, y, z, \ldots$. For any particular application, the alphabet is assumed fixed and all definitions are relative to the alphabet.

Types are built up from the set of type constructors and the set of parameters, using the symbols $\rightarrow$ and $\times$.

Definition 2.1.2. A *type* is defined inductively as follows.

1. Each parameter in $\mathfrak{P}$ is a type.
2. If T is a type constructor in $\mathfrak{T}$ of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (For $k = 0$, this reduces to a type constructor of arity 0 being a type.)
3. If α and β are types, then $\alpha \rightarrow \beta$ is a type.
4. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type. (For $n = 0$, this reduces to *1* being a type.)

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

$\mathfrak{S}$ denotes the set of all types obtained from an alphabet ($\mathfrak{S}$ for 'sort'). The symbol $\to$ is right associative, so that $\alpha \to \beta \to \gamma$ means $\alpha \to (\beta \to \gamma)$.

It is worthwhile being precise about the exact meaning of the phrase 'defined inductively' in Definition 2.1.2. First, a universe of expressions is needed. In this case, the set of all finite sequences of symbols drawn from the set $\mathfrak{T}$ of type constructors, the set $\mathfrak{P}$ of parameters, and '$\to$' and '$\times$' will serve. Then the definition states that $\mathfrak{S}$ is the intersection of all sets $\mathfrak{X}$ of expressions such that the following conditions are satisfied.

1. $\mathfrak{P} \subseteq \mathfrak{X}$.
2. If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.
3. If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \to \beta \in \mathfrak{X}$.
4. If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.

There is always at least one such set satisfying these conditions, namely the set of all expressions. Thus the intersection is well defined and, furthermore, it satisfies Conditions 1 to 4, as can easily be checked. Hence $\mathfrak{S}$ is the smallest set of expressions satisfying Conditions 1 to 4 (where one set is 'smaller' than another if the former is a subset of the latter). Corresponding to this definition, there is also the following *principle of induction on the structure of types.*

Proposition 2.1.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{S}$ satisfying the following conditions.*

1. $\mathfrak{P} \subseteq \mathfrak{X}$.
2. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.*
3. *If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \to \beta \in \mathfrak{X}$.*
4. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{S}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 4 of the definition of a type and $\mathfrak{S}$ is the intersection of all such sets, it follows immediately that $\mathfrak{S} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{S}$. □

The assumptions $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ in Condition 2 of Proposition 2.1.1 are collectively known as the *induction hypothesis* (for that step of the proof). Similarly for Conditions 3 and 4. The majority of the proofs in this book are induction arguments on the structure of various kinds of terms or types. The previous discussion, illustrated by the case of types, makes clear the precise basis of these arguments.

Definition 2.1.3. A type is *closed* if it contains no parameters.

Notation 2.1.1. $\mathfrak{S}^c$ denotes the set of all closed types obtained from an alphabet.

Note that $\mathfrak{S}^c$ is non-empty, since 1, $\Omega \in \mathfrak{S}^c$. The next result provides a useful characterisation of $\mathfrak{S}^c$.

Proposition 2.1.2. *$\mathfrak{S}^c$ is the intersection of all sets $\mathfrak{Y}$ of types such that the following hold.*

1. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{Y}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$.*
2. *If α, $\beta \in \mathfrak{Y}$, then $\alpha \to \beta \in \mathfrak{Y}$.*
3. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{Y}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{Y}$.*

Proof. First, $\mathfrak{S}^c$ is a set of types that satisfies Conditions 1, 2, and 3. Hence $\bigcap \mathfrak{Y} \subseteq \mathfrak{S}^c$.

On the other hand, let $\mathfrak{Y}$ be a set of types that satisfies Conditions 1, 2, and 3. I show that $\mathfrak{S}^c \subseteq \mathfrak{Y}$. Let $\mathfrak{X} = \{\alpha \in \mathfrak{S} \mid \alpha$ is closed implies that $\alpha \in \mathfrak{Y}\}$. It suffices to show that $\mathfrak{X} = \mathfrak{S}$. The proof is by induction on the structure of types.

If a is a parameter, then $a \in \mathfrak{X}$ (since a is not closed).

Suppose that $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k. If there exists $j \in \{1, \ldots, k\}$ such that α_j is not closed, then $T\ \alpha_1 \ldots \alpha_k$ is not closed and hence $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$. Otherwise, α_i is closed and so $\alpha_i \in \mathfrak{Y}$, for $i = 1, \ldots, k$. Then, according to Condition 1, $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$. Thus $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.

Suppose that $\alpha, \beta \in \mathfrak{X}$. If one or both of α or β is not closed, then $\alpha \to \beta$ is not closed and hence $\alpha \to \beta \in \mathfrak{X}$. Otherwise, α and β are closed and so $\alpha, \beta \in \mathfrak{Y}$. Then, according to Condition 2, $\alpha \to \beta \in \mathfrak{Y}$. Thus $\alpha \to \beta \in \mathfrak{X}$.

Suppose that $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$. If there exists $j \in \{1, \ldots, n\}$ such that α_j is not closed, then $\alpha_1 \times \cdots \times \alpha_n$ is not closed and hence $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$. Otherwise, α_i is closed and so $\alpha_i \in \mathfrak{Y}$, for $i = 1, \ldots, n$. Then, according to Condition 3, $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{Y}$. Thus $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.

All four conditions in Proposition 2.1.1 have now been shown to hold, so that $\mathfrak{X} = \mathfrak{S}$, by the principle of induction on the structure of types. Thus $\mathfrak{S}^c \subseteq \mathfrak{Y}$. Since this is true for all such $\mathfrak{Y}$, it follows that $\mathfrak{S}^c \subseteq \bigcap \mathfrak{Y}$. Thus $\mathfrak{S}^c = \bigcap \mathfrak{Y}$. □

Using Proposition 2.1.2, the following *principle of induction on the structure of closed types* can be proved.

Proposition 2.1.3. *Let $\mathfrak{X}$ be a subset of $\mathfrak{S}^c$ satisfying the following conditions.*

1. *If $\alpha_1, \ldots, \alpha_k \in \mathfrak{X}$ and T is a type constructor in $\mathfrak{T}$ of arity k, then $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{X}$.*
2. *If $\alpha, \beta \in \mathfrak{X}$, then $\alpha \to \beta \in \mathfrak{X}$.*
3. *If $\alpha_1, \ldots, \alpha_n \in \mathfrak{X}$, then $\alpha_1 \times \cdots \times \alpha_n \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{S}^c$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of Proposition 2.1.2 and $\mathfrak{S}^c$ is the intersection of all such sets, it follows immediately that $\mathfrak{S}^c \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{S}^c$. □

Proposition 2.1.2 provides a 'top-down' characterisation of $\mathfrak{S}^c$. There is also a 'bottom-up' characterisation. Let $\mathbb{N}$ denote the set of natural numbers, that is, non-negative integers.

Definition 2.1.4. Define $\{\mathfrak{S}^c_m\}_{m \in \mathbb{N}}$ inductively as follows.

$$\begin{aligned}
\mathfrak{S}^c_0 &= \{T \mid T \in \mathfrak{T} \text{ has arity } 0\} \\
\mathfrak{S}^c_{m+1} &= \{T\ \alpha_1 \ldots \alpha_k \mid T \in \mathfrak{T} \text{ has arity } k, \alpha_i \in \mathfrak{S}^c_m (1 \leq i \leq k), k \in \mathbb{N}\} \\
&\quad \cup \{\alpha \to \beta \mid \alpha, \beta \in \mathfrak{S}^c_m\} \\
&\quad \cup \{\alpha_1 \times \cdots \times \alpha_n \mid \alpha_i \in \mathfrak{S}^c_m (1 \leq i \leq n), n \in \mathbb{N}\}.
\end{aligned}$$

Proposition 2.1.4.

1. $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c_{m+1}$, *for* $m \in \mathbb{N}$.
2. $\mathfrak{S}^c = \bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m$.

Proof. 1. This is an easy induction argument.

2. First, I show that $\bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m \subseteq \mathfrak{S}^c$. To prove this, it suffices to show by induction that $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c$, for $m \in \mathbb{N}$. Clearly, $\mathfrak{S}^c_0 \subseteq \mathfrak{S}^c$. Suppose next that $\mathfrak{S}^c_m \subseteq \mathfrak{S}^c$. It then follows from the definition of $\mathfrak{S}^c_{m+1}$ and the fact that $\mathfrak{S}^c$ satisfies Conditions 1, 2, and 3 in Proposition 2.1.2 that $\mathfrak{S}^c_{m+1} \subseteq \mathfrak{S}^c$.

Now I show that $\mathfrak{S}^c \subseteq \bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m$. Put $\bigcup_{m \in \mathbb{N}} \mathfrak{S}^c_m = \mathfrak{Y}$. It suffices to show that $\mathfrak{Y}$ satisfies Conditions 1, 2, and 3 of Proposition 2.1.2. Suppose that $\alpha_1, \ldots, \alpha_k \in \mathfrak{Y}$ and T is a type constructor in $\mathfrak{T}$ of arity k. Since the $\mathfrak{S}^c_m$ are increasing, there exists $p \in \mathbb{N}$ such that $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c_p$. Hence $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{S}^c_{p+1}$ and so $T\ \alpha_1 \ldots \alpha_k \in \mathfrak{Y}$. Similar arguments show that $\mathfrak{Y}$ satisfies Conditions 2 and 3. □

Proposition 2.1.5. *If* $\mathfrak{T}$ *is countable, then* $\mathfrak{S}^c$ *is countable.*

Proof. By Proposition 2.1.4, it suffices to show by induction that each $\mathfrak{S}^c_m$ is countable. Clearly, $\mathfrak{S}^c_0$ is countable. Suppose now that $\mathfrak{S}^c_m$ is countable. By the definition of $\mathfrak{S}^c_{m+1}$ and the fact that $\mathfrak{T}$ is countable, it follows easily that $\mathfrak{S}^c_{m+1}$ is also countable. □

Example 2.1.1. In practical applications of the logic, a variety of types is needed. For example, declarative programming languages typically admit the following types (which are nullary type constructors): *1*, *Ω*, *Nat* (the type of natural numbers), *Int* (the type of integers), *Float* (the type of floating-point numbers), *Real* (the type of real numbers), *Char* (the type of characters), and *String* (the type of strings).

Other useful type constructors are those used to define lists, trees, and so on. In the logic, *List* denotes the (unary) list type constructor. Thus, if α is a type, then *List* α is the type of lists whose elements have type α.

2.2 Type Substitutions

Fundamental to the later definition of a term, and elsewhere, is the concept of a type substitution unifying a set of equations about types. To explore this, note first that types are essentially first-order terms. A type constructor of arity n is essentially a (first-order) function symbol of arity n. Similarly, one can think of $\rightarrow$ as a binary function symbol and $\times \cdots \times$ (where there are n occurrences of $\times$) as an $(n+1)$-ary function symbol. Thus the unification problem for types is the same as that encountered for terms in first-order logic. For this reason, the standard development of (first-order) unification is not repeated here, but some definitions and facts are merely recalled for later use, and the reader is left to consult standard references on the topic, if more detail is required.

Definition 2.2.1. A *type substitution* is a finite set of the form $\{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$, where each a_i is a parameter, each α_i is a type distinct from a_i, and $a_1, \ldots, a_n$ are distinct. Each element a_i/α_i is called a *binding.*

In particular, $\{\}$ is a type substitution called the *identity substitution.*

Notation 2.2.1. If $\mu = \{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$, then $domain(\mu) = \{a_1, \ldots, a_n\}$ and $range(\mu)$ is the set of parameters appearing in $\{\alpha_1, \ldots, \alpha_n\}$.

Definition 2.2.2. A type substitution $\{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$ is *closed* if α_i is closed, for $i = 1, \ldots, n$.

Definition 2.2.3. Let $\mu = \{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$ be a type substitution and α a type. Then $\alpha\mu$, the *instance* of α by μ, is the type obtained from α by simultaneously replacing each occurrence of the parameter a_i in α by the type α_i $(i = 1, \ldots, n)$.

Definition 2.2.4. Let $\mu = \{a_1/\alpha_1, \ldots, a_m/\alpha_m\}$ and $\nu = \{b_1/\beta_1, \ldots, b_n/\beta_n\}$ be type substitutions. Then the *composition* $\mu\nu$ of μ and ν is the type substitution obtained from the set

$$\{a_1/\alpha_1\nu, \ldots, a_m/\alpha_m\nu, b_1/\beta_1, \ldots, b_n/\beta_n\}$$

by deleting any binding $a_i/\alpha_i\nu$ for which $a_i = \alpha_i\nu$ and deleting any binding b_j/β_j for which $b_j \in \{a_1, \ldots, a_m\}$.

Composition is defined so that $\alpha(\mu\nu) = (\alpha\mu)\nu$, for any type α and type substitutions μ and ν. (See Exercise 2.3.) Also $\mu\{\} = \{\}\mu = \mu$, for any μ, so $\{\}$ really is an identity. Composition of type substitutions is associative.

One type can be more general than another.

Definition 2.2.5. Let α and β be types. Then α is *more general than* β if there exists a type substitution ξ such that $\beta = \alpha\xi$.

Note that 'more general than' includes 'equal to', since ξ can be the identity substitution.

Example 2.2.1. Let $\alpha = (List\ a) \times \Omega$ and $\beta = (List\ Int) \times \Omega$. Then α is more general than β, since $\beta = \alpha\xi$, where $\xi = \{a/Int\}$.

Definition 2.2.6. Let $E = \{\alpha_1 = \beta_1, \ldots, \alpha_n = \beta_n\}$ be a set of equations about types. Then a type substitution μ is a *unifier* for E if $\alpha_i\mu$ is identical to $\beta_i\mu$, for $i = 1, \ldots, n$.

Example 2.2.2. Let $E = \{a \to List\ b = c \to List\ M, a \times b = a \times M\}$, where a, b and c are parameters and M is a unary type constructor. Then the type substitution $\{a/M, b/M, c/M\}$ is a unifier for E.

One type substitution can be more general than another.

Definition 2.2.7. Let μ and ν be type substitutions. Then μ is *more general* than ν if there exists a type substitution γ such that $\mu\gamma = \nu$.

Definition 2.2.8. Let E be a set of equations about types and μ be a unifier for E. Then μ is a *most general unifier* for E if, for every unifier ν for E, μ is more general than ν.

Notation 2.2.2. The phrase 'most general unifier' is often abbreviated to 'mgu'.

Example 2.2.3. Let $E = \{a \to List\ b = c \to List\ M, a \times b = a \times M\}$ be a set of equations about types. Then the type substitution $\{a/c, b/M\}$ is an mgu for E. Note that, for the unifier $\{a/M, b/M, c/M\}$ of Example 2.2.2, $\{a/M, b/M, c/M\} = \{a/c, b/M\}\{c/M\}$. Thus $\{a/c, b/M\}$ is indeed more general than this unifier.

A general unifier of a set of equations is unique 'modulo renaming of parameters'. To make this statement precise, invertible type substitutions are defined.

Definition 2.2.9. A type substitution μ is *invertible* if there exists a type substitution μ^{-1} such that $\mu\mu^{-1} = \mu^{-1}\mu = \{\}$. A type substitution μ^{-1} satisfying this condition is called an *inverse* of μ.

If a type substitution has an inverse, then the inverse is unique, as can easily be established.

Definition 2.2.10. A type substitution $\mu = \{a_1/b_1, \ldots, a_n/b_n\}$ is a *permutation of parameters* if the b_is are distinct parameters and $domain(\mu) = range(\mu)$.

It can be shown that a type substitution is invertible iff it is a permutation of parameters.

Now the precise statement about the essential uniqueness of mgus can be given. Let μ_1 and μ_2 be mgus. Then μ_1 and μ_2 are mgus of the same set of equations about types iff $\mu_1 = \mu_2\alpha$, for some invertible type substitution α.

The next definition introduces some useful terminology.

Definition 2.2.11. A type σ is a *variant* of type τ if $\sigma = \tau\alpha$, for some invertible type substitution α.

There are various algorithms for unifying a set of equations about first-order terms and, therefore, about types. The main fact is that unification is decidable, that is, given a set of equations about types, either the set is unifiable and the algorithm returns an mgu or the set is not unifiable and the algorithm reports this fact. These algorithms typically return an mgu that involves only the parameters appearing in the set of equations. (Of course, many mgus involve new parameters, since an mgu can be composed with an invertible type substitution containing new parameters to give another mgu.) Thus, if an equation set is unifiable, it can be supposed without loss of generality that there is an mgu involving only the parameters appearing in the equation set.

For later use, two results about type substitutions are now established.

Proposition 2.2.1. *Let E and F be sets of equations about types.*

1. *If σ is an mgu for E and μ an mgu for $F\sigma$, then $\sigma\mu$ is an mgu for $E \cup F$.*
2. *Let $E \cup F$ have mgu δ. Then there exist type substitutions σ and μ such that σ is an mgu for E, the parameters in σ all appear in E, μ is an mgu for $F\sigma$, and $\delta = \sigma\mu$.*
3. *If $E \cup F$ is unifiable and σ is an mgu for E, then $F\sigma$ is unifiable.*

Proof. 1. Since $(E \cup F)\sigma\mu = E\sigma\mu \cup F\sigma\mu$, $\sigma\mu$ is a unifier for $E \cup F$. Suppose that ξ is a unifier for $E \cup F$. Thus ξ is a unifier for E and so $\xi = \sigma\alpha$, for some α. Also $\sigma\alpha$ is a unifier for F and so α is a unifier for $F\sigma$. Since μ is an mgu for $F\sigma$, it follows that $\alpha = \mu\beta$, for some β. Hence $\xi = \sigma\mu\beta$ and thus $\sigma\mu$ is an mgu for $E \cup F$.

2. Since δ is a unifier for E, E has an mgu σ, where $\delta = \sigma\xi$, for some ξ. It can be assumed without loss of generality that all the parameters in σ appear in E. Thus $F\sigma$ is unifiable because $F\sigma\xi = F\delta$. Let μ' be an mgu for $F\sigma$. By Part 1, $\sigma\mu'$ is an mgu for $E \cup F$. Since δ is also an mgu for $E \cup F$, it follows that $\delta = \sigma\mu'\alpha$, for some invertible type substitution α. Put $\mu = \mu'\alpha$. Then $\delta = \sigma\mu$, where σ is an mgu for E and μ is an mgu for $F\sigma$.

3. Let γ be a unifier for $E \cup F$. Thus γ unifies E. Since σ is an mgu for E, $\gamma = \sigma\delta$, for some δ. Thus δ unifies $F\sigma$, since $(F\sigma)\delta = F\gamma$. □

Proposition 2.2.2. *Let $E_1, \ldots, E_n$ and F be sets of equations about types, where the parameters in each E_i are disjoint from one another. Suppose that*

$E_1 \cup \cdots \cup E_n \cup F$ has mgu δ. Then there exist type substitutions $\sigma_1, \ldots, \sigma_n$ and μ such that σ_i is an mgu for E_i, the parameters in σ_i all appear in E_i $(i = 1, \ldots, n)$, μ is an mgu for $F(\sigma_1 \cup \cdots \cup \sigma_n)$, and $\delta = (\sigma_1 \cup \cdots \cup \sigma_n)\mu$.

Proof. The proof is by induction on n.

If $n = 0$, the result is obvious with $\mu = \delta$.

For the induction step, consider the set of equations $E_1 \cup \cdots \cup E_n \cup F$. By the induction hypothesis, there exist type substitutions $\sigma_1, \ldots, \sigma_{n-1}$ and μ' such that σ_i is an mgu for E_i, the parameters in σ_i all appear in E_i $(i = 1, \ldots, n-1)$, μ' is an mgu for $(E_n \cup F)(\sigma_1 \cup \cdots \cup \sigma_{n-1})$, and $\delta = (\sigma_1 \cup \cdots \cup \sigma_{n-1})\mu'$. Note that $(E_n \cup F)(\sigma_1 \cup \cdots \cup \sigma_{n-1}) = E_n \cup F(\sigma_1 \cup \cdots \cup \sigma_{n-1})$.

By Part 2 of Proposition 2.2.1, there exist type substitutions σ_n and μ such that σ_n is an mgu for E_n, the parameters in σ_n all appear in E_n, μ is an mgu for $F(\sigma_1 \cup \cdots \cup \sigma_{n-1})\sigma_n$ and $\mu' = \sigma_n \mu$. Since each of the σ_i contain distinct sets of parameters, $(\sigma_1 \cup \cdots \cup \sigma_{n-1})\sigma_n = \sigma_1 \cup \cdots \cup \sigma_n$. Thus $F(\sigma_1 \cup \cdots \cup \sigma_{n-1})\sigma_n = F(\sigma_1 \cup \cdots \cup \sigma_n)$. Also $\delta = (\sigma_1 \cup \cdots \cup \sigma_{n-1})\mu' = (\sigma_1 \cup \cdots \cup \sigma_{n-1})\sigma_n \mu = (\sigma_1 \cup \cdots \cup \sigma_n)\mu$. □

2.3 Terms

Definition 2.3.1. A *signature* is the declared type for a constant.

Notation 2.3.1. The fact that a constant C has signature α is often denoted by $C : \alpha$.

Two different kinds of constants, *data constructors* and *functions*, are distinguished. In a knowledge representation context, data constructors are used to represent individuals. In a programming language context, data constructors are used to construct data values. In contrast, functions are used to compute on data values; functions have definitions while data constructors do not. In the semantics for the logic, the data constructors are used to construct models. As examples, the constants $\top$ (true) and $\bot$ (false) are data constructors, as is each integer, floating-point number, and character. The constant $\sharp$ (cons) used to construct lists is a data constructor. The constants *append* and *concat* introduced in examples below are functions.

A *predicate* is a function having signature of the form $\alpha \rightarrow \Omega$, for some α. Predicates will play an important role in the application of the logic to learning.

The set $\mathfrak{C}$ always includes the following constants (where a is a parameter).

1. $()$, having signature 1.
2. $=$, having signature $a \rightarrow a \rightarrow \Omega$.
3. $\top$ and $\bot$, having signature Ω.
4. $\neg$, having signature $\Omega \rightarrow \Omega$.
5. $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$, having signature $\Omega \rightarrow \Omega \rightarrow \Omega$.

6. Σ and Π, having signature $(a \rightarrow \Omega) \rightarrow \Omega$.

The intended meaning of $=$ is identity (that is, $= x\ y$ is $\top$ iff x and y are identical), the intended meaning of $\top$ is true, the intended meaning of $\bot$ is false, and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$ are as usual. The intended meanings of Σ and Π are that Σ maps a predicate to $\top$ iff the predicate maps at least one element to $\top$ and Π maps a predicate to $\top$ iff the predicate maps all elements to $\top$.

Note 2.3.1. In this book, the equality symbol '$=$' is overloaded. On the one hand, '$=$' is a constant in the alphabet of a higher-order logic. On the other hand, '$=$' is a symbol of the informal meta-language in which the book is written with the intended meaning of identity. The meaning of any occurrence of the symbol '$=$' will always be clear from the context. Equality is nearly always written infix.

Data constructors always have a signature of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$, where T is a type constructor of arity k, $a_1, \ldots, a_k$ are distinct parameters, and all the parameters appearing in $\sigma_1, \ldots, \sigma_n$ occur among $a_1, \ldots, a_k$ $(n \geq 0,\ k \geq 0)$. The *arity* of the data constructor is n. A *nullary* data constructor is a data constructor of arity 0. The arity of a data constructor C is denoted by $arity(C)$.

Example 2.3.1. The data constructors for constructing lists are $[]$ having signature $List\ a$ and $\sharp$ having signature $a \rightarrow List\ a \rightarrow List\ a$, where $\sharp$ is usually written infix. $[]$ represents the empty list. The term $s \sharp t$ represents the list with head s and tail t. Thus $4 \sharp 5 \sharp 6 \sharp []$ represents the list $[4, 5, 6]$. In fact, $[t_1, \ldots, t_n]$ is often used as notational sugar for the term $t_1 \sharp \cdots \sharp t_n \sharp []$.

It is assumed throughout that, for each type constructor T, there exists at least one data constructor having a signature of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$. If C is a data constructor having a signature of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$, then C is said to be *associated with* the type constructor T.

The next task is to define the central concept of a term. In the non-polymorphic case, a simple inductive definition suffices. But the polymorphic case is more complicated since, when putting terms together to make larger terms, it is generally necessary to solve a system of equations and these equations depend upon the relative types of free variables in the component terms. The effect of this is that to define a term one has to define simultaneously its type, and its set of free variables and their relative types.

Definition 2.3.2. A *term*, together with its type, and its set of free variables and their relative types, is defined inductively as follows.

1. Each variable x in $\mathfrak{V}$ is a term of type a, where a is a parameter. The variable x is free with relative type a in x.

2. Each constant C in $\mathfrak{C}$, where C has signature α, is a term of type α.
3. (Abstraction) If t is a term of type β and x a variable in $\mathfrak{V}$, then $\lambda x.t$ is a term of type $\alpha \to \beta$, if x is free with relative type α in t, or type $a \to \beta$, where a is a new parameter, otherwise.
   A variable other than x is free with relative type σ in $\lambda x.t$ if the variable is free with relative type σ in t.
4. (Application) If s is a term of type α and t a term of type β such that the equation
   $$\alpha = \beta \to b,$$
   where b is a new parameter, augmented with equations of the form
   $$\varrho = \delta,$$
   for each variable that is free with relative type ϱ in s and is also free with relative type δ in t, have a most general unifier ξ, then $(s\ t)$ is a term of type $b\xi$.
   A variable is free with relative type $\sigma\xi$ in $(s\ t)$ if the variable is free with relative type σ in s or t.
5. (Tuple) If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, such that the set of equations of the form
   $$\varrho_{i_1} = \cdots = \varrho_{i_k},$$
   for each variable that is free with relative type ϱ_{i_j} in the term t_{i_j} ($j = 1, \ldots, k$ and $k > 1$), have a most general unifier ξ, then $(t_1, \ldots, t_n)$ is a term of type $\alpha_1\xi \times \cdots \times \alpha_n\xi$.
   A variable is free with relative type $\sigma\xi$ in $(t_1, \ldots, t_n)$ if the variable is free with relative type σ in t_j, for some $j \in \{1, \ldots, n\}$.

The type substitution ξ in Parts 4 and 5 of the definition is called the *associated* mgu.

Notation 2.3.2. $\mathfrak{L}$ denotes the set of all terms obtained from an alphabet and is called the *language* given by the alphabet.

In Parts 4 and 5 of Definition 2.3.2, it is understood that parameters in types in the respective component terms are standardised apart (that is, renamed to avoid undesirable name clashes). More precisely, in Part 4, it is understood that the parameters in α and the relative types of the free variables in s are standardised apart from the parameters in β and the relative types of the free variables in t. The standardisation apart can be achieved by applying a suitable invertible type substitution to one of the sets of types. Similarly, in Part 5, it is understood that the respective sets of parameters in each α_i and the relative types of free variables in t_i are standardised apart from one another.

In Part 4 of Definition 2.3.2, the typical case is when s is a term whose type has the form $\gamma \to \varepsilon$. In this case, the first equation can be replaced by

$$\gamma = \beta$$

and the type of $(s\ t)$ will be $\varepsilon\xi$. However, it is also possible for the type of s to be a parameter, for example, if s is a variable, and this is the reason for the particular formulation of the equations in that part.

In Part 5, if $n = 1$, (t_1) is defined to be t_1. If $n = 0$, the term obtained is the empty tuple, $()$, which is a term of type *1*.

The set of equations in Part 4 of Definition 2.3.2 that have to have a unifier in order to form an application are denoted by $Constraints_{(s\ t)}$. Thus $Constraints_{(s\ t)}$ is

$$\{\alpha = \beta \to b\} \cup$$
$$\{\varrho = \delta \mid \text{ there is a variable that is free with relative type } \varrho \text{ in } s \text{ and free with relative type } \delta \text{ in } t\}.$$

Similarly, the equations in Part 5 are denoted by $Constraints_{(t_1,\dots,t_n)}$. Thus $Constraints_{(t_1,\dots,t_n)}$ is

$$\{\varrho_{i_1} = \cdots = \varrho_{i_k} \mid \text{ there is a variable that is free with relative type } \varrho_{i_j} \text{ in the term } t_{i_j} (j = 1,\dots,k \text{ and } k > 1)\}.$$

The type of a term is not unique because there may be many mgus of the corresponding set of equations. However, all these mgus differ by just an invertible type substitution and thus the corresponding types differ by just a renaming of parameters.

Definition 2.3.3. A term is *closed* if it contains no free variables.

Definition 2.3.4. A term of type Ω is called a *formula.*

Definition 2.3.5. A *theory* is a collection of formulas.

Note 2.3.2. Since Definition 2.3.2 is rather complicated, some remarks to clarify its exact meaning are in order. First, rather more than just a term is being defined. In fact, what shall be referred to as an *annotated term*, which is a triple, is actually being defined. The first component is the term itself, the second is its type, and the third is its set of free variables and their relative types. The inductive definition specifies that the set of annotated terms is the intersection of all sets of triples satisfying (appropriately reformulated versions of) Parts 1 to 5 of the definition (which are then called Conditions 1 to 5).

For this definition to make sense, one needs a suitable universe of triples. For a place in a triple where a type or relative type should appear, one can

use a type as defined earlier. For the place where a term should appear, one can define a suitable set of expressions as follows. Given some alphabet, let an expression be a finite sequence of symbols drawn from the set of constants $\mathfrak{C}$, the set of variables $\mathfrak{V}$, and '(', ')', 'λ', '.', and ','. Then the universe is the set of triples for which the first component is an expression, the second is a type, and the third is a set of pairs consisting of a variable and a type. Thus the intersection of all sets of triples satisfying Conditions 1 to 5 is well defined and, furthermore, it satisfies Conditions 1 to 5. Hence the set of annotated terms is the smallest set of triples satisfying Conditions 1 to 5.

Generally, when proving properties of terms, what are actually used are the corresponding annotated terms since there is usually some reference in the proof to the type of the term or to free variables and their relative types. When needed, this will be taken for granted in the following without any remark to this effect being made. In particular, where appropriate, $\mathfrak{L}$ will implicitly refer to the set of annotated terms.

In Part 4 of Definition 2.3.2, a variable can be free in both s and t. To determine the relative type of the variable in $(s\ t)$, one can use either the relative type ϱ in s or the relative type δ in t. The reason, of course, is that ξ is a unifier of the set of equations for this part and so $\varrho\xi = \delta\xi$. A similar remark applies to Part 5.

To prove properties of terms, one can employ the following *principle of induction on the structure of terms.*

Proposition 2.3.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{L}$ satisfying the following conditions.*

1. $\mathfrak{V} \subseteq \mathfrak{X}$.
2. $\mathfrak{C} \subseteq \mathfrak{X}$.
3. *If $t \in \mathfrak{X}$ and $x \in \mathfrak{V}$, then $\lambda x.t \in \mathfrak{X}$.*
4. *If $s, t \in \mathfrak{X}$ and $(s\ t) \in \mathfrak{L}$, then $(s\ t) \in \mathfrak{X}$.*
5. *If $t_1, \ldots, t_n \in \mathfrak{X}$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{L}$.

Proof. The first step is to show that $\mathfrak{X}$ satisfies Conditions 1 to 5 of the definition of a term. For Conditions 1, 2, and 3, this is immediate from the first three conditions satisfied by $\mathfrak{X}$.

For Condition 4 of the definition of a term, suppose that $s, t \in \mathfrak{X}$ and $Constraints_{(s\ t)}$ is unifiable. Since $\mathfrak{L}$ satisfies Condition 4, it follows that $(s\ t) \in \mathfrak{L}$. Thus $(s\ t) \in \mathfrak{X}$, by the fourth condition satisfied by $\mathfrak{X}$. Hence $\mathfrak{X}$ satisfies Condition 4 of the definition of a term.

For Condition 5 of the definition of a term, the proof is similar.

Now, since $\mathfrak{X}$ satisfies Conditions 1 to 5 of the definition of a term and $\mathfrak{L}$ is the intersection of all such sets, it follows immediately that $\mathfrak{L} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{L}$. □

Since $\mathfrak{L}$ is a subset of the set of all strings over the alphabet consisting of the set of variables, the set of constants, 'λ', and the punctuation symbols, and since the set of strings over any alphabet is well founded under the substring relation $\prec$ (Example A.1.1), it follows that $\mathfrak{L}$ is well founded under the same relation. As a well-founded set, there is a principle of induction (Proposition A.1.3) available for proving properties of terms and a principle of inductive construction (Proposition A.1.4) available for defining functions whose domain is a set of terms. Note that the minimal elements of $\mathfrak{L}$ under $\prec$ are the constants and variables.

Here are three examples to illustrate Definition 2.3.2.

Example 2.3.2. Let M be a nullary type, and $A : M$ and $concat : List\ a \times List\ a \rightarrow List\ a$ be constants. Recall that $[] : List\ a$ and $\sharp : a \rightarrow List\ a \rightarrow List\ a$ are the data constructors for lists. I will show that $(concat\ ([], [A]))$ is a term. For this, $([], [A])$ must be shown to be a term, which leads to the consideration of $[]$ and $[A]$. Now $[]$ is a term of type $List\ a$, by Part 2 of the definition of a term. By Parts 2 and 4, $(\sharp\ A)$ is a term of type $List\ M \rightarrow List\ M$, where along the way the equation $a = M$ is solved with the mgu $\{a/M\}$. Then $((\sharp\ A)\ [])$, which is the list $[A]$, is a term of type $List\ M$ by Part 4, where the equation $List\ M = List\ a$ is solved. By Part 5, it follows that $([], [A])$ is a term of type $List\ a \times List\ M$. Finally, by Part 4 again, $(concat\ ([], [A]))$ is a term of type $List\ M$, where the equation to be solved is $List\ a \times List\ a = List\ a \times List\ M$ whose mgu is $\{a/M\}$.

Example 2.3.3. Consider the constants $append : List\ a \rightarrow List\ a \rightarrow List\ a \rightarrow \Omega$ and $process : List\ a \rightarrow List\ a$. I will show that $(((append\ x)\ [])\ (process\ x))$ is a term. First, the variable x is a term of type b, where the parameter is chosen to avoid a clash in the next step. Then $(append\ x)$ is a term of type $List\ a \rightarrow List\ a \rightarrow \Omega$, for which the equation solved is $List\ a = b$. Next $((append\ x)\ [])$ is a term of type $List\ a \rightarrow \Omega$ and x has relative type $List\ a$ in $((append\ x)\ [])$. Now consider $(process\ x)$, for which the constituent parts are $process$ of type $List\ c \rightarrow List\ c$ and the variable x of type d. Thus $(process\ x)$ is a term of type $List\ c$ and x has relative type $List\ c$ in $(process\ x)$. Finally, one has to apply $((append\ x)\ [])$ to the term $(process\ x)$. For this, by Part 4, there are two equations. These are $List\ a = List\ c$, coming from the top-level types, and $List\ a = List\ c$, coming from the free variable x in each of the components. These equations have the mgu $\{c/a\}$. Thus $(((append\ x)\ [])\ (process\ x))$ is a term of type Ω.

Example 2.3.4. If x is a variable (of type a) and y is a variable (of type b), then $(x\ y)$ is a term of type c, where c is a new parameter, and x has relative type $b \rightarrow c$ in $(x\ y)$.

Notation 2.3.3. Terms of the form $(\Sigma\ \lambda x.t)$ are written as $\exists x.t$ and terms of the form $(\Pi\ \lambda x.t)$ are written as $\forall x.t$ (in accord with the intended meaning of Σ and Π). In a higher-order logic, one may identify sets and predicates

– the actual identification is between a set and its characteristic function which is a predicate. Thus, if t is of type Ω, the abstraction $\lambda x.t$ may be written as $\{x \mid t\}$ if it is intended to emphasise that its intended meaning is a set. The notation $\{\}$ means $\{x \mid \bot\}$. The notation $s \in t$ means $(t\ s)$, where t has type $\mu \rightarrow \Omega$, for some μ. Furthermore, notwithstanding the fact that sets are mathematically identified with predicates, it is sometimes convenient to maintain an informal distinction between sets (as 'collections of objects') and predicates. For this reason, the notation $\{\mu\}$ is introduced as a synonym for the type $\mu \rightarrow \Omega$. The term $(s\ t)$ is often written as simply $s\ t$, using juxtaposition to denote application. Juxtaposition is left associative, so that $r\ s\ t$ means $((r\ s)\ t)$. Thus $(((append\ x)\ [])\ (process\ x))$ can be written more simply as $append\ x\ []\ (process\ x)$.

Proposition 2.3.2. *Let $t \in \mathfrak{L}$. Then exactly one of the following conditions holds.*

1. $t \in \mathfrak{V}$.
2. $t \in \mathfrak{C}$.
3. *t has the form $\lambda x.s$, where $s \in \mathfrak{L}$.*
4. *t has the form $(u\ v)$, where $u, v \in \mathfrak{L}$.*
5. *t has the form $(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n \in \mathfrak{L}$.*

Proof. It is clear that at most one of the conditions holds. Now suppose t is a term that is neither a variable, nor a constant, nor has the form $\lambda x.s$, $(u\ v)$, or $(t_1, \ldots, t_n)$. Then $\mathfrak{L} \setminus \{t\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Thus t is either a variable, a constant, or has the form $\lambda x.s$, $(u\ v)$, or $(t_1, \ldots, t_n)$.

Suppose that t has the form $\lambda x.s$, but that s is not a term. Then the set of terms $\mathfrak{L} \setminus \{t\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Thus s is a term. The arguments for Parts 4 and 5 are similar. □

Proposition 2.3.3.

1. *An expression of the form $\lambda x.t$ is a term iff t is a term.*
2. *An expression of the form $(s\ t)$ is a term iff s and t are terms and $Constraints_{(s\ t)}$ is unifiable.*
3. *An expression of the form $(t_1, \ldots, t_n)$ is a term iff $t_1, \ldots, t_n$ are terms and $Constraints_{(t_1,\ldots,t_n)}$ is unifiable.*

Proof. 1. If $\lambda x.t$ is a term, then t is a term by Part 3 of Proposition 2.3.2. Conversely, if t is a term, then $\lambda x.t$ is a term, since $\mathfrak{L}$ satisfies Condition 3 of the definition of a term.

2. Suppose that $(s\ t)$ is a term. Then s and t are terms by Part 4 of Proposition 2.3.2. If $Constraints_{(s\ t)}$ is not unifiable, then $\mathfrak{L} \setminus \{(s\ t)\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition

of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Conversely, if s and t are terms and $Constraints_{(s\ t)}$ is unifiable, then $(s\ t)$ is a term, since $\mathfrak{L}$ satisfies Condition 4 of the definition of a term.

3. Suppose that $(t_1, \ldots, t_n)$ is a term. Then $t_1, \ldots, t_n$ are terms by Part 5 of Proposition 2.3.2. If $Constraints_{(t_1,\ldots,t_n)}$ is not unifiable, then $\mathfrak{L} \setminus \{(t_1, \ldots, t_n)\}$ satisfies Conditions 1 to 5 in the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. Conversely, if $t_1, \ldots, t_n$ are terms and $Constraints_{(t_1,\ldots,t_n)}$ is unifiable, then $(t_1, \ldots, t_n)$ is a term, since $\mathfrak{L}$ satisfies Condition 5 of the definition of a term. □

A common task, especially in the programming language applications of the logic, is to check that putative terms are correctly typed. A suitable class of putative terms is given by the set of well-formed expressions defined as follows.

Definition 2.3.6. A *well-formed expression* is defined inductively as follows.

1. Each variable in $\mathfrak{V}$ is a well-formed expression.
2. Each constant in $\mathfrak{C}$ is a well-formed expression.
3. If e is a well-formed expression and x is a variable, then $\lambda x.e$ is a well-formed expression.
4. If d and e are well-formed expression, then $(d\ e)$ is a well-formed expression.
5. If $e_1, \ldots, e_n$ are well-formed expressions, then $(e_1, \ldots, e_n)$ is a well-formed expression.

Thus the well-formed expressions are the terms of the untyped λ-calculus. Clearly the set of well-formed expressions (on some alphabet) is generally much larger than the set of terms (that is, $\mathfrak{L}$). The type checking problem is to determine whether or not some given well-formed expression t is in $\mathfrak{L}$ and, if so, to determine the type of t.

Proposition 2.3.4. *Type checking of terms is decidable and each well-formed expression can be well-typed in at most one way (up to variants).*

Proof. The proof proceeds by induction on the structure of well-formed expressions. □

2.4 Subterms

The concept of a subterm of a term will play an important part in the applications of the logic. As preparation for the definition of a subterm, the concept of an occurrence in a term is defined. Let $\mathbb{Z}$ denote the set of integers, $\mathbb{Z}^+$ the set of positive integers, and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string.

Definition 2.4.1. The *occurrence set* of a term t, denoted $\mathcal{O}(t)$, is the set of strings in $(\mathbb{Z}^+)^*$ defined inductively as follows.

1. If t is a variable, then $\mathcal{O}(t) = \{\varepsilon\}$.
2. If t is a constant, then $\mathcal{O}(t) = \{\varepsilon\}$.
3. If t has the form $\lambda x.s$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \{1o \mid o \in \mathcal{O}(s)\}$.
4. If t has the form $(u\ v)$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \{1o \mid o \in \mathcal{O}(u)\} \cup \{2o' \mid o' \in \mathcal{O}(v)\}$.
5. If t has the form $(t_1, \ldots, t_n)$, then $\mathcal{O}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{io_i \mid o_i \in \mathcal{O}(t_i)\}$.

Each $o \in \mathcal{O}(t)$ is called an *occurrence* in t.

More precisely, $\mathcal{O}$ is a function $\mathcal{O} : \mathfrak{L} \to 2^{(\mathbb{Z}^+)^*}$ from the set of terms into the powerset of the set of all strings of positive integers. The existence and uniqueness of $\mathcal{O}$ depends upon that fact that $\mathfrak{L}$ is well founded under the substring relation and hence Proposition A.1.4 applies: $\mathcal{O}$ is defined directly on the minimal elements (that is, constants and variables) and is uniquely determined by the rules in the definition for abstractions, applications, and tuples.

Example 2.4.1. Consider the term *append* x [] (*process* x), illustrated in Fig. 2.1 below. Its occurrence set is $\{\varepsilon, 1, 2, 11, 12, 21, 22, 111, 112\}$.

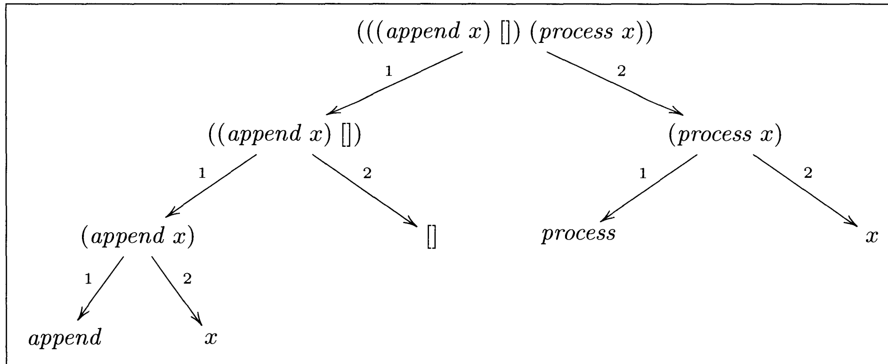

Fig. 2.1. A term with occurrences

Definition 2.4.2. If t is a term and $o \in \mathcal{O}(t)$, then the *subterm of t at occurrence o*, denoted $t|_o$, is defined inductively on the length of o as follows.

1. If $o = \varepsilon$, then $t|_o = t$.
2. If $o = 1o'$, for some o', and t has the form $\lambda x.s$, then $t|_o = s|_{o'}$.
   If $o = 1o'$, for some o', and t has the form $(u\ v)$, then $t|_o = u|_{o'}$.
   If $o = 2o'$, for some o', and t has the form $(u\ v)$, then $t|_o = v|_{o'}$.
   If $o = io'$, for some o', and t has the form $(t_1, \ldots, t_n)$, then $t|_o = t_i|_{o'}$, for $i = 1, \ldots, n$.

A *subterm* is a subterm of a term at some occurrence. A subterm is *proper* if it is not at occurrence ε.

In connection with Definition 2.4.2, note that if t has the form $\lambda x.s$, then s is a term, by Proposition 2.3.2. Thus $s|_{o'}$ is well defined. Similar comments apply to $(u\ v)$ and $(t_1, \ldots, t_n)$. Furthermore, if t is a variable or a constant, then $\mathcal{O}(t)$ must be $\{\varepsilon\}$. Also if t has the form $\lambda x.s$, then each $o \in \mathcal{O}(t)$ must be either ε or have the form $1o'$, for some o'. Similar comments apply to $(u\ v)$ and $(t_1, \ldots, t_n)$.

Example 2.4.2. Consider the term *append* x [] (*process* x). The subterm of *append* x [] (*process* x) at occurrence

ε is *append* x [] (*process* x),

1 is *append* x [],

2 is *process* x,

11 is *append* x,

12 is [],

21 is *process*,

22 is x,

111 is *append*, and

112 is x.

These subterms are illustrated in Fig. 2.1.

Proposition 2.4.1. *Each subterm is a term.*

Proof. Let t be a term and $o \in \mathcal{O}(t)$. It is shown by induction on the length of o that $t|_o$ is a term.

If the length of o is 0, then $o = \varepsilon$. Thus $t|_o = t$, which is a term.

For the inductive step, suppose the length of o is $n+1$ ($n \geq 0$). There are several cases to consider.

If $o = 1o'$, for some o', and t has the form $\lambda x.s$, then $t|_o = s|_{o'}$ which is a term by the induction hypothesis.

If $o = 1o'$, for some o', and t has the form $(u\ v)$, then $t|_o = u|_{o'}$ which is a term by the induction hypothesis.

If $o = 2o'$, for some o', and t has the form $(u\ v)$, then $t|_o = v|_{o'}$ which is a term by the induction hypothesis.

If $o = io'$, for some o', and t has the form $(t_1, \ldots, t_n)$, then $t|_o = t_i|_{o'}$ which is a term by the induction hypothesis, for $i = 1, \ldots, n$. □

If r is a subterm of s at occurrence o' and s is a subterm of t at occurrence o, then r is a subterm of t at occurrence oo', where oo' denotes the concatenation of o and o'. Thus a subterm of a subterm of a term t is a subterm of t.

Suppose for some term, s is a subterm at occurrence o and s' is a subterm at occurrence o'. Then either o is a prefix of o', in which case s' is a subterm of s, or o' is a prefix of o, in which case s is a subterm of s', or neither o nor o' is a prefix of the other, in which case s and s' do not overlap.

Definition 2.4.3. Two occurrences o and o' are *disjoint* if neither o nor o' is a prefix of the other. A set of occurrences $\{o_i\}_{i=1}^n$ is *disjoint* if each pair of occurrences in the set is disjoint.

Two subterms of a term are *disjoint* if their occurrences are disjoint. A set of subterms of a term is *disjoint* if their set of occurrences is disjoint.

Proposition 2.4.2.

1. *Each constant appearing at a particular place in a term is a subterm of the term.*
2. *Each variable appearing at a particular place in a term (except a variable appearing immediately after a λ) is a subterm of the term.*

Proof. 1. The proof is by induction on the structure of terms. Let t be a term and C a constant appearing in t. It has to be shown that there exists $o \in \mathcal{O}(t)$ such that $t|_o = C$.

If t is a variable, then there can be no appearance of a constant and the result holds.

If t is the constant C, then the only appearance of a constant is t itself and $t|_\varepsilon = C$.

Let t have the form $\lambda x.s$ and C be a constant appearing in t. Thus C must appear in s. By the induction hypothesis, there exists $o' \in \mathcal{O}(s)$ such that $s|_{o'} = C$. Thus $t|_{1o'} = C$.

The argument when t has the form $(u\ v)$ or $(t_1, \ldots, t_n)$ is similar.

2. The proof is by induction on the structure of terms. Let t be a term and x a variable appearing (not immediately after a λ) in t. It has to be shown that there exists $o \in \mathcal{O}(t)$ such that $t|_o = x$.

If t is a variable x, then the only appearance of a variable is t itself and $t|_\varepsilon = x$.

If t is a constant, then there can be no appearance of a variable and the result holds.

Let t have the form $\lambda y.s$ and x be a variable appearing in t. Since x does not appear immediately after a λ, x must appear in s. By the induction hypothesis, there exists $o' \in \mathcal{O}(s)$ such that $s|_{o'} = x$. Thus $t|_{1o'} = x$.

The argument when t has the form $(u\ v)$ or $(t_1, \ldots, t_n)$ is similar. □

Note that a variable appearing immediately after a λ in a term is *not* a subterm since the variable appearing there is not at an occurrence of the term.

Definition 2.4.4. An occurrence of a variable x in a term is *bound* if it occurs within a subterm of the form $\lambda x.t$.

A variable in a term is *bound* if it has a bound occurrence.

An occurrence of a variable in a term is *free* if it is not a bound occurrence.

For a particular occurrence of a subterm $\lambda x.t$ in a term, the occurrence of t is called the *scope* of the λx.

In the definition of a term, the concept of a free variable is also defined. The following result establishes that free variables are exactly those variables which have free occurrences.

Proposition 2.4.3. *A variable is free in a term iff it has a free occurrence in the term.*

Proof. The proof is by induction on the structure of terms. Let the term be t.

If t is a variable or a constant, then the result is clear.

Let t have the form $\lambda x.s$. I claim that x is not a free variable in t. Thus suppose that x is a free variable in t. Then $\mathfrak{L} \setminus \{t\}$ satisfies Conditions 1 to 5 of the definition of a term, which contradicts the definition of $\mathfrak{L}$ as being the smallest set satisfying Conditions 1 to 5. (If x is a free variable for the (annotated) term t, then t is not needed for Condition 3 to be satisfied.) Thus x is not a free variable in t. Furthermore, any occurrence of x in t is bound, by definition. Thus the property holds for x. For another variable y occurring in s, by the induction hypothesis, y is free in s iff y has a free occurrence in s. Thus y is free in t iff y has a free occurrence in t.

Let t have the form $(u\ v)$. Then a variable x is free in $(u\ v)$ iff x is free in either u or v iff x has free occurrence in either u or v (by the induction hypothesis) iff x has a free occurrence in $(u\ v)$.

Let t have the form $(t_1, \ldots, t_n)$. Then a variable x is free in $(t_1, \ldots, t_n)$ iff x is free in t_j, for some $j \in \{1, \ldots, n\}$ iff x has a free occurrence in t_j, for some $j \in \{1, \ldots, n\}$ (by the induction hypothesis) iff x has a free occurrence in $(t_1, \ldots, t_n)$. □

A variable can be bound by more than one λ in a term.

Example 2.4.3. Let M and N be nullary types, and $f : M \to \Omega$ and $g : N \to \Omega$ be constants. Then $(\Sigma\ \lambda x.(f\ x)) \wedge (\Sigma\ \lambda x.(g\ x))$ is a term of type Ω, in which the variable x is bound by several λs.

A variable can be both bound and free in a term.

Example 2.4.4. Let L, M, N, and P be nullary type constructors, and $f : (L \to M) \to N \to P$, $g : L \to M$, and $h : L \to L \to N$ be constants. Consider the term $f\ \lambda x.(g\ x)\ (h\ x\ y)$. Then the first occurrence of x (that is, the x in $(g\ x)$) is bound and the second occurrence is free. Thus x is both bound and free in $f\ \lambda x.(g\ x)\ (h\ x\ y)$.

The last two examples motivate the introduction of the following concept.

Definition 2.4.5. A term is *rectified* if each bound variable is bound by just one λ and no variable is both bound and free.

By a suitable change of bound variables any term can be converted to a rectified term. Since the names of the bound variables are of no consequence, the rectified term thus obtained is 'equivalent' to the original term. Later, terms that differ only in the names of the bound variables are identified.

The concept of relative type of free variables is now extended to all subterms of a term.

Definition 2.4.6. The *relative type* of the subterm $t|_o$ of the term t at $o \in \mathcal{O}(t)$ is defined by induction on the length of o as follows.

If the length of o is 0, then the relative type of $t|_o$ is the same as the type of t.

For the inductive step, suppose the length of o is $n+1$ ($n \geq 0$). There are several cases to consider.

If $o = 1o'$, for some o', and t has the form $\lambda x.s$, then the relative type of $t|_o$ is the same as the relative type of $s|_{o'}$ in s.

If $o = 1o'$, for some o', and t has the form $(u\ v)$, then the relative type of $t|_o$ is $\sigma\xi$, where σ is the relative type of $u|_{o'}$ in u and ξ is the associated mgu for $(u\ v)$.

If $o = 2o'$, for some o', and t has the form $(u\ v)$, then the relative type of $t|_o$ is $\sigma\xi$, where σ is the relative type of $v|_{o'}$ in v and ξ is the associated mgu for $(u\ v)$.

If $o = io'$, for some $i \in \{1, \ldots, n\}$ and o', and t has the form $(t_1, \ldots, t_n)$, then the relative type of $t|_o$ is $\sigma\xi$, where σ is the relative type of $t_i|_{o'}$ in t_i and ξ is the associated mgu for $(t_1, \ldots, t_n)$.

Note 2.4.1. Having given the definition of relative type of a subterm, a rather subtle clarification of the definition is needed concerning the use of the associated mgus. In the definition of the concept of a term in which associated mgus were also defined, it was only required that the sets of parameters in the types of the constituent terms together with the types of the free variables in these terms be standardised apart. For Definition 2.4.6 and in what follows, a stronger form of standardisation apart is needed because it is possible for subterms of a term to contain parameters in their relative types that do not occur amongst the parameters in the type of the term or the types of free variables. This can happen if there is a function that has a signature of the form $\alpha \to \beta$, where α contains a parameter that does not appear in β. (Such functions are common – see Chap. 5.) For example, if $f : a \to M$ is a function, then $(f\ [])$ is a term of type M, whereas the relative type of $[]$ in $(f\ [])$ is *List b*. Thus it is understood in Definition 2.4.6 that before an associated mgu is formed the respective sets of parameters in the *relative types of the subterms* of the constituent terms are standardised apart.

This stronger form of standardisation apart cannot be imposed in the definition of a term because there is no concept of a subterm at that stage,

nor is it needed. However, it is possible to extend the definition of term to simultaneously include the definitions of subterm and relative type of a subterm and, in this case, the stronger form of standardisation apart would be needed.

Example 2.4.5. Let M be a nullary type, and $append : List\ a \rightarrow List\ a \rightarrow List\ a \rightarrow \Omega$, $process : List\ a \rightarrow List\ a$ and $A, B, C : M$ be constants. Consider the first occurrence of x in the term $append\ x\ []\ (process\ x)$. As a term in its own right, x has type a, for some parameter a. As a subterm of $append\ x\ []\ (process\ x)$, x has relative type $List\ a$.

The occurrence of x in the term $append\ x\ []\ (process\ [A, B, C])$ has relative type $List\ M$. Also the subterm $process$ of this term has relative type $List\ M \rightarrow List\ M$.

In the definition of a term, the concept of the relative type of a free variable is defined. The next proposition shows that this relative type is the same as the relative type of any of the free occurrences of the variable.

Proposition 2.4.4. *The relative type of each free variable x in a term t is the same as the relative type of each free occurrence of x in t.*

Proof. The proof is by induction on the structure of terms.

Let t be the variable x of type a. The relative type of x as a free variable is a and the relative type of the free occurrence of x in x is also a.

If t is a constant, there are no free variables and the result holds.

Let t have the form $\lambda y.s$ and x be a free variable in t. Then x cannot be y and x must be a free variable in s. By the induction hypothesis, the relative type of x in s is the same as the relative types of each of its free occurrences in s. Thus the relative type of x in t is the same as the relative types of each of its free occurrences in t.

Let t have the form $(u\ v)$ and x be a free variable in t. Thus x is a free variable in either u or v or both. Choose any free occurrence of x in u, say. By the induction hypothesis, the relative type of x in u is the same as the relative type of this free occurrence of x in u. Thus the relative type of x in $(u\ v)$ is the same as the relative type of this free occurrence of x in $(u\ v)$.

Let t have the form $(t_1, \ldots, t_n)$ and x be a free variable in t. Thus x is a free variable in t_j, for some $j \in \{1, \ldots, n\}$. Choose any free occurrence of x in t_j. By the induction hypothesis, the relative type of x in t_j is the same as the relative type of this free occurrence of x in t_j. Thus the relative type of x in $(t_1, \ldots, t_n)$ is the same as the relative type of this free occurrence of x in $(t_1, \ldots, t_n)$. □

Distinct bound occurrences of a variable can have distinct relative types.

Example 2.4.6. Let M and N be nullary types, and $f : M \rightarrow \Omega$ and $g : N \rightarrow \Omega$ be constants, and t the term $(\Sigma\ \lambda x.(f\ x)) \wedge (\Sigma\ \lambda x.(g\ x))$. The bound occurrence of x in $(\Sigma\ \lambda x.(f\ x))$ has relative type M in t, while the one in $(\Sigma\ \lambda x.(g\ x))$ has relative type N in t.

The next topic is concerned with replacing a subterm of a term by a new subterm.

Definition 2.4.7. Let t be a term, s a subterm of t at occurrence o, and r a term. Then the expression obtained by *replacing* s *in* t *by* r, denoted $t[s/r]_o$, is defined by induction on the length of o as follows.

If the length of o is 0, then $t[s/r]_o = r$.

For the inductive step, suppose the length of o is $n+1$ ($n \geq 0$). There are several cases to consider.

If $o = 1o'$, for some o', and t has the form $\lambda x.w$, then $(\lambda x.w)[s/r]_o = \lambda x.(w[s/r]_{o'})$.

If $o = 1o'$, for some o', and t has the form $(u\ v)$, then $(u\ v)[s/r]_o = (u[s/r]_{o'}\ v)$.

If $o = 2o'$, for some o', and t has the form $(u\ v)$, then $(u\ v)[s/r]_o = (u\ v[s/r]_{o'})$.

If $o = io'$, for some $i \in \{1, \ldots, n\}$ and o', and t has the form $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)[s/r]_o = (t_1, \ldots, t_i[s/r]_{o'}, \ldots, t_n)$.

More generally, one can replace several subterms simultaneously.

Definition 2.4.8. Let t be a term, s_i a subterm of t at occurrence o_i, and r_i a term, for $i = 1, \ldots, n$, such that the set of subterms $\{s_i\}_{i=1}^n$ is disjoint. Then the expression obtained by *replacing* s_i *in* t *by* r_i, *for* $i = 1, \ldots, n$, denoted $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$, is defined by induction on the number n of subterms by $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n} = (t[s_1/r_1]_{o_1})[s_2/r_2, \ldots, s_n/r_n]_{o_2,\ldots,o_n}$.

Since the subterms are disjoint, it is clear that $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$ is well defined (because the occurrences continue to be well defined as the successive replacements are made) and that the result of the replacements does not depend on the order in which they are carried out. When there is no chance of confusion, the specific mention of the occurrences may be dropped and $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$ denoted more simply by $t[s_1/r_1, \ldots, s_n/r_n]$.

Easy examples show that $t[s/r]_o$ does not have to be a term (because r may not have the correct type). However, under certain conditions that arise particularly in the application of the logic to declarative programming languages, it will be a term.

Definition 2.4.9. Let s be a term of type σ and t a term of type τ. Then s is *type-weaker* than t, denoted $s \precsim t$, if there exists a type substitution γ such that $\tau = \sigma\gamma$, every free variable in s is a free variable in t, and, if the relative type of a free variable in s is δ, then the relative type of this free variable in t is $\delta\gamma$.

Notation 2.4.1. The type substitution γ in Definition 2.4.9 is denoted by $Subst_{s \precsim t}$.

Example 2.4.7. Let $s = y$ and $t = f\ x\ y$, where $f : M \to N \to N$. Let $\gamma = \{b/N\}$, where b is the type of y. Then $b\gamma = N$ and s is type-weaker than t.

Example 2.4.8. Let $s = y$ and $t = f\ y\ x$, where $f : M \to N \to N$. Then s is not type-weaker than t since no suitable γ exists.

The next proposition collects some basic properties of $\precsim$.

Proposition 2.4.5.

1. *Let t be a term. Then $t \precsim t$.*
2. *Let r, s, and t be terms. If $r \precsim s$ and $s \precsim t$, then $r \precsim t$.*
3. *Let s and t be terms. If $s \precsim t$, then $\lambda x.s \precsim \lambda x.t$.*
4. *Let s_1, s_2, t_1, and t_2 be terms and $(t_1\ t_2)$ a term. If $s_1 \precsim t_1$ and $s_2 \precsim t_2$, then $(s_1\ s_2)$ is a term and $(s_1\ s_2) \precsim (t_1\ t_2)$.*
5. *Let s_i and t_i be terms, for $i = 1, \ldots, n$, and $(t_1, \ldots, t_n)$ a term. If $s_i \precsim t_i$, for $i = 1, \ldots, n$, then $(s_1, \ldots, s_n)$ is a term and $(s_1, \ldots, s_n) \precsim (t_1, \ldots, t_n)$.*

Proof. 1. This part is obvious.

2. This part is obvious.

3. Suppose that s has type σ, and $Subst_{s \precsim t} = \gamma$. There are three cases.

(a) If x is free with relative type α in s, then the type of $\lambda x.s$ is $\alpha \to \sigma$. Thus the relative type of x in t is $\alpha\gamma$, so that the type of $\lambda x.t$ is $(\alpha \to \sigma)\gamma$. If y is a free variable of relative type δ in $\lambda x.s$, then y has relative type $\delta\gamma$ in $\lambda x.t$. Thus $Subst_{\lambda x.s \precsim \lambda x.t} = \gamma$.

(b) If x is not free in s, but is free with relative type β in t, then the type of $\lambda x.s$ is $a \to \sigma$, where a is a new parameter, and the type of $\lambda x.t$ is $\beta \to \sigma\gamma = (a \to \sigma)\gamma'$, where $\gamma' = \{a/\beta\} \cup \gamma$. If y is a free variable of relative type δ in $\lambda x.s$, then y has relative type $\delta\gamma = \delta\gamma'$ in $\lambda x.t$. Thus $Subst_{\lambda x.s \precsim \lambda x.t} = \gamma'$.

(c) If x is not free in both s and t, then the type of $\lambda x.s$ is $a \to \sigma$, where a is a new parameter, the type of $\lambda x.t$ is $b \to \sigma\gamma$, where b is a new parameter, and $b \to \sigma\gamma = (a \to \sigma)\gamma''$, where $\gamma'' = \{a/b\} \cup \gamma$. If y is a free variable of relative type δ in $\lambda x.s$, then y has relative type $\delta\gamma = \delta\gamma''$ in $\lambda x.t$. Thus $Subst_{\lambda x.s \precsim \lambda x.t} = \gamma''$.

4. Suppose that s_1 has type μ_1, t_1 has type η_1, s_2 has type μ_2, t_2 has type η_2, and $Subst_{s_i \precsim t_i} = \gamma_i$, for $i = 1, 2$. It can be supposed without loss of generality that γ_1 applies only to parameters appearing in μ_1 and relative types of free variables in s_1, γ_2 applies only to parameters appearing in μ_2 and relative types of free variables in s_2, and that the γ_i have no parameters in common. Suppose that β is an mgu for $Constraints_{(t_1\ t_2)}$ that includes the equation $\eta_1 = \eta_2 \to b$. Clearly $(\gamma_1 \cup \gamma_2)\beta$ is a unifier for $Constraints_{(s_1\ s_2)}$ that includes the equation $\mu_1 = \mu_2 \to b$ (where without loss of generality the new parameter b is the same as the new parameter in $Constraints_{(t_1\ t_2)}$). Let

α be an mgu for *Constraints*$_{(s_1\ s_2)}$. Thus $(\gamma_1 \cup \gamma_2)\beta = \alpha\pi$, for some π. Also $(s_1\ s_2)$ is a term of type $b\alpha$ and, if x is a free variable of relative type δ in some s_i, then x has relative type $\delta\alpha$ in $(s_1\ s_2)$.

Now $b\alpha\pi = b(\gamma_1 \cup \gamma_2)\beta = b\beta$, where $b\beta$ is the type of $(t_1\ t_2)$. Finally, let x be a free variable of type ε in $(s_1\ s_2)$. Thus x has relative type δ in some s_i and $\varepsilon = \delta\alpha$. Now $\varepsilon\pi = \delta\alpha\pi = \delta(\gamma_1 \cup \gamma_2)\beta = \delta\gamma_i\beta$, where $\delta\gamma_i\beta$ is the relative type of x in $(t_1\ t_2)$. Thus $(s_1\ s_2) \precsim (t_1\ t_2)$ and *Subst*$_{(s_1\ s_2)\precsim(t_1\ t_2)} = \pi$.

5. Suppose that s_i has type σ_i, t_i has type τ_i, and *Subst*$_{s_i \precsim t_i} = \gamma_i$, for $i = 1,\ldots,n$. It can be supposed without loss of generality that γ_i applies only to parameters appearing in σ_i and relative types of free variables in s_i, for $i = 1,\ldots,n$, and that the γ_i have no parameters in common. Suppose that β is an mgu for *Constraints*$_{(t_1,\ldots,t_n)}$. Clearly $(\gamma_1 \cup \cdots \cup \gamma_n)\beta$ is a unifier for *Constraints*$_{(s_1,\ldots,s_n)}$. Let α be an mgu for *Constraints*$_{(s_1,\ldots,s_n)}$. Thus $(\gamma_1 \cup \cdots \cup \gamma_n)\beta = \alpha\pi$, for some π. Also $(s_1,\ldots,s_n)$ is a term of type $(\sigma_1 \times \cdots \times \sigma_n)\alpha$ and, if x is a free variable of relative type δ in some s_i, then x has relative type $\delta\alpha$ in $(s_1,\ldots,s_n)$.

Now $(\sigma_1 \times \cdots \times \sigma_n)\alpha\pi = (\sigma_1 \times \cdots \times \sigma_n)(\gamma_1 \cup \cdots \cup \gamma_n)\beta = (\tau_1 \times \cdots \times \tau_n)\beta$, where $(\tau_1 \times \cdots \times \tau_n)\beta$ is the type of $(t_1,\ldots,t_n)$. Finally, let x be a free variable of type ε in $(s_1,\ldots,s_n)$. Thus x has relative type δ in some s_i and $\varepsilon = \delta\alpha$. Now $\varepsilon\pi = \delta\alpha\pi = \delta(\gamma_1 \cup \cdots \cup \gamma_n)\beta = \delta\gamma_i\beta$, where $\delta\gamma_i\beta$ is the relative type of x in $(t_1,\ldots,t_n)$. Thus $(s_1,\ldots,s_n) \precsim (t_1,\ldots,t_n)$ and *Subst*$_{(s_1,\ldots,s_n)\precsim(t_1,\ldots,t_n)} = \pi$.

□

Proposition 2.4.6 below is concerned with a particular situation in which replacing a subterm of a term by another term gives a term again. This result will be needed later to establish some properties of computations.

Proposition 2.4.6. *Let t be a term, s a subterm of t at occurrence o, and r a term such that $r \precsim s$. Then the following hold.*

1. *$t[s/r]_o$ is a term and $t[s/r]_o \precsim t$.*
2. *$t = t[s/r]_o$ is a term.*

Proof. 1. The proof is by induction on the length of o.

If the length of o is 0, then $s = t$, $t[s/r]_o = r$, and $t[s/r]_o$ is type-weaker than t.

For the inductive step, suppose the length of o is $n+1$ $(n \geq 0)$. There are several cases to consider.

If $o = 1o'$, for some o', and t has the form $\lambda x.w$, then $(\lambda x.w)[s/r]_o = \lambda x.(w[s/r]_{o'})$. By the induction hypothesis, $w[s/r]_{o'}$ is a term that is type-weaker than w. Thus $\lambda x.(w[s/r]_{o'})$ is a term that is type-weaker than $\lambda x.w$, by Part 3 of Proposition 2.4.5. That is, $t[s/r]_o$ is a term that is type-weaker than t.

If $o = 1o'$, for some o', and t has the form $(u\ v)$, then $(u\ v)[s/r]_o = (u[s/r]_{o'}\ v)$. By the induction hypothesis, $u[s/r]_{o'}$ is a term that is type-weaker than u. Hence $(u[s/r]_{o'}\ v)$ is a term that is type-weaker than $(u\ v)$,

by Part 4 of Proposition 2.4.5. That is, $t[s/r]_o$ is a term that is type-weaker than t.

If $o = 2o'$, for some o', and t has the form $(u\ v)$, then the argument is similar to the previous part.

If $o = io'$, for some $i \in \{1, \ldots, n\}$ and o', and t has the form $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)[s/r]_o = (t_1, \ldots, t_i[s/r]_{o'}, \ldots, t_n)$. By the induction hypothesis, $t_i[s/r]_{o'}$ is a term that is type-weaker than t_i. Hence $(t_1, \ldots, t_i[s/r]_q, \ldots, t_n)$ is a term that is type-weaker than $(t_1, \ldots, t_n)$, by Part 5 of Proposition 2.4.5. That is, $t[s/r]_o$ is a term that is type-weaker than t.

2. This part follows easily from the facts that $=$ has signature $a \to a \to \Omega$ and that $t[s/r]_o$ is a term that is type-weaker than t. □

Example 2.4.9. Let $t = \lambda x.(f\ x\ y)$, where $f : M \to N \to N$. Thus t has type $M \to N$. Let $s = f\ x\ y$ (the subterm of t at occurrence 1), $r = y$, and $\gamma = \{b/N\}$, where b is the type of y. Then $b\gamma = N$ and $r \precsim s$. Now $t[s/r]_1 = \lambda x.y$, which has type $a \to b$, for some new parameter a. Let $\xi = \{a/M, b/N\}$. Then $(a \to b)\xi = M \to N$ and $t[s/r]_1 \precsim t$.

Definition 2.4.10. Two terms s and t are *type-equivalent*, denoted $s \approx t$, if they have the same types, the same set of free variables, and, for every free variable x in s and t, x has the same relative type in s as it has in t (up to variants).

Proposition 2.4.7. *Let t be a term, s a subterm of t at occurrence o, and r a term such that $r \approx s$. Then $t[s/r]_o$ is a term and $t[s/r]_o \approx t$.*

Proof. The proof follows immediately from Proposition 2.4.6 by considering $t[s/r]_o$ obtained from t by replacing s by r, and t obtained from $t[s/r]_o$ by replacing r by s. □

2.5 Term Substitutions

In this section, the concept of instantiating a term by a substitution is studied.

Definition 2.5.1. A *term substitution* is a finite set of the form $\{x_1/t_1, \ldots, x_n/t_n\}$, where each x_i is a variable, each t_i is a term distinct from x_i, and $x_1, \ldots, x_n$ are distinct. Each element x_i/t_i is called a *binding.*

In particular, $\{\}$ is a term substitution called the *identity substitution.*

Notation 2.5.1. If $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, then $domain(\theta) = \{x_1, \ldots, x_n\}$ and $range(\theta)$ is the set of free variables appearing in $\{t_1, \ldots, t_n\}$.

Notation 2.5.2. Let θ be a term substitution and t a term. Then $\theta|_t$ is the term substitution obtained from θ by restricting θ to just the free variables appearing in t.

Definition 2.5.2. A term substitution θ is *idempotent* if $domain(\theta) \cap range(\theta) = \emptyset$.

The usual definition of an idempotent substitution θ is that $\theta\theta = \theta$. However, this relies on having a definition of composition of term substitutions, which is problematical because of the need to avoid free variable capture (see below). In fact, while composition of type substitutions is needed in many places in this book, composition of term substitutions is never needed, so the concept is eschewed altogether. It will be easy to arrange for all necessary term substitutions in later developments to be idempotent.

Intuitively, the concept of instantiating a term t by a term substitution $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$, is simple – each free occurrence of a variable x_i in t is replaced by t_i. But there is a technical complication in that there may be a free variable y, say, in some t_i that is 'captured' in this process because, after instantiation, it occurs in the scope of a subterm of the form $\lambda y.s$ and therefore becomes bound in $t\theta$. Free variable capture spoils the intended meaning of instantiation and hence it is necessary to avoid it. There are two approaches to this: one can disallow instantiation if free variable capture would occur or one can rename bound variables in the term t to avoid free variable capture altogether. The latter approach is adopted here.

Definition 2.5.3. Let t be a term and $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ a term substitution. The *instance* $t\theta$ of t by θ is the well-formed expression defined as follows.

1. If t is a variable x_i, for some $i \in \{1, \ldots, n\}$, then $x_i\theta = t_i$.
   If t is a variable y distinct from all the x_i, then $y\theta = y$.
2. If t is a constant C, then $C\theta = C$.
3. If t is an abstraction $\lambda x_i.s$, for some $i \in \{1, \ldots, n\}$, then

   $$(\lambda x_i.s)\theta = \lambda x_i.(s\{x_1/t_1, \ldots, x_{i-1}/t_{i-1}, x_{i+1}/t_{i+1}, \ldots, x_n/t_n\}).$$

   If t is an abstraction $\lambda y.s$, where y is distinct from all the x_i, and, for all $i \in \{1, \ldots, n\}$, either y is not free in t_i or x_i is not free in s, then

   $$(\lambda y.s)\theta = \lambda y.(s\theta).$$

   If t is an abstraction $\lambda y.s$, where y is distinct from all the x_i, and, for some $i \in \{1, \ldots, n\}$, y is free in t_i and x_i is free in s, then

   $$(\lambda y.s)\theta = \lambda z.(s\{y/z\}\theta).$$

   (Here z is chosen to be the first variable that is not free in s or any of the t_i.)
4. If t is an application $(u\ v)$, then $(u\ v)\theta = (u\theta\ v\theta)$.
5. If t is a tuple $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)\theta = (t_1\theta, \ldots, t_n\theta)$.

Note 2.5.1. In later proofs, it is assumed that the parameters in the type of t_i and relative types of its free variables are standardised apart from the parameters in the type of t_j and relative types of its free variables ($i, j = 1, \ldots, n$ and $i \neq j$), and also that the parameters that appear in the types of the t_i and the relative types of their free variables are standardised apart from those in the type of the term and the relative types of its free variables to which the term substitution is applied.

Example 2.5.1. Suppose that t is the term $append\ (u, v, w)$ and θ is the term substitution $\{u/1 \sharp x, v/[], w/[1,2]\}$. Then $t\theta = append\ (1 \sharp x, [], [1,2])$.

Example 2.5.2. Suppose that t is the term $\exists r.\exists x.\exists y.(u = r \sharp x \wedge w = r \sharp y \wedge append\ (x, v, y))$ and θ is the term substitution $\{u/1 \sharp x, v/[], w/[1,2]\}$. Then

$$t\theta = \exists r.\exists z.\exists y.(1 \sharp x = r \sharp z \ \wedge\ [1,2] = r \sharp y \ \wedge\ append\ (z, [], y)).$$

Here the bound variable x in t is renamed to z to avoid capture of the x in the binding $u/1 \sharp x$.

Proposition 2.5.1. *Let t be a term and $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ a term substitution, where x_i is not free in t, for $i = 1, \ldots, n$. Then $t\theta = t$.*

Proof. The proof is by induction on the structure of t.

If t is a variable y, then $y \neq x_i$, for $i = 1, \ldots, n$, and so $y\theta = y$.

If t is a constant C, then $C\theta = C$.

Let t be an abstraction $\lambda y.s$. Suppose first that $y = x_i$, for some i. Then $(\lambda y.s)\theta = \lambda y.(s\{x_1/t_1, \ldots, x_{i-1}/t_{i-1}, x_{i+1}/t_{i+1}, \ldots, x_n/t_n\}) = \lambda y.s$, by the induction hypothesis. Suppose now y is distinct from all the x_i. Then no x_i is free in s and so $(\lambda y.s)\theta = \lambda y.(s\theta) = \lambda y.s$, by the induction hypothesis.

If t is an application $(u\ v)$, then $(u\ v)\theta = (u\theta\ v\theta) = (u\ v)$, by the induction hypothesis.

It t is a tuple $(t_1, \ldots, t_n)$, then $(t_1, \ldots, t_n)\theta = (t_1\theta, \ldots, t_n\theta) = (t_1, \ldots, t_n)$, by the induction hypothesis. □

Note that $t\theta$ may not be a term.

Example 2.5.3. Let M and N be nullary type constructors and $F : M \rightarrow \Omega$ and $A : N$ be constants. Let t be $(F\ x)$ and θ be $\{x/A\}$. Then $t\theta = (F\ A)$ which is not a term.

The following result provides a condition for $t\theta$ to be a term. First, some useful notation for this is given.

Notation 2.5.3. Let t be a term having type σ and $x_1, \ldots, x_n$ be free variables in t, where the relative type of x_i in t is ϱ_i ($i = 1, \ldots, n$). Let $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ be an idempotent term substitution, where t_i has type τ_i ($i = 1, \ldots, n$). Then

$$
\begin{aligned}
U_{t,\theta} &= \{\varrho_i = \tau_i \mid i = 1, \ldots, n\}, \text{ and} \\
V_{t,\theta} &= \{\delta_{i_1} = \cdots = \delta_{i_k} [= \delta] \mid \text{ there is a variable that is free with relative type } \delta_{i_j} \text{ in } t_{i_j} \text{ and, possibly, the variable is free with relative type } \delta \text{ in } t\}.
\end{aligned}
$$

Proposition 2.5.2. *Let t be a term having type σ and $x_1, \ldots, x_n$ variables that each occur freely exactly once in t. Let $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ be an idempotent term substitution. If $U_{t,\theta} \cup V_{t,\theta}$ has mgu φ, then $t\theta$ is a term of type $\sigma\varphi$ and, if y is a free variable in $t\theta$ and y has relative type δ in t or some t_i, then y has relative type $\delta\varphi$ in $t\theta$.*

Proof. The proof proceeds by induction on the structure of terms.

In case t is a variable or constant, the result is obvious.

Suppose t has the form $\lambda x.s$ with type $\alpha \rightarrow \beta$. Then

$U_{\lambda x.s,\theta} \cup V_{\lambda x.s,\theta}$ has mgu φ

implies $U_{s,\theta} \cup V_{s,\theta}$ has mgu φ

[since $U_{s,\theta} = U_{\lambda x.s,\theta}$ and $V_{s,\theta} = V_{\lambda x.s,\theta}$. It can be assumed without loss of generality that $x \notin range(\theta)$]

implies $s\theta$ is a term of type $\beta\varphi$ and, if y is a free variable in $s\theta$ and y has relative type δ in s or some t_i, then y has relative type $\delta\varphi$ in $s\theta$

[by the induction hypothesis]

implies $(\lambda x.s)\theta$ is a term of type $(\alpha \rightarrow \beta)\varphi$ and, if y is a free variable in $(\lambda x.s)\theta$ and y has relative type δ in s or some t_i, then y has relative type $\delta\varphi$ in $(\lambda x.s)\theta$.

[It suffices to show that $(\lambda x.s)\theta$ is a term of type $(\alpha \rightarrow \beta)\varphi$. If x is not free in s, then α is a parameter a and $(\lambda x.s)\theta$ has type $a \rightarrow \beta\varphi = (a \rightarrow \beta)\varphi$. If x is free in s, then x has relative type α in s so that $(\lambda x.s)\theta$ has type $(\alpha \rightarrow \beta)\varphi$]

Suppose t has the form $(u_1\ u_2)$ with type σ, where u_1 has type α, u_2 has type β, and μ is the associated mgu for $(u_1\ u_2)$. Hence $\sigma = b\mu$, where $\alpha = \beta \rightarrow b$. Then

$U_{(u_1\ u_2),\theta} \cup V_{(u_1\ u_2),\theta}$ has mgu φ

implies $U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup U_{u_2,\theta|_{u_2}} \cup V_{u_2,\theta|_{u_2}} \cup X \cup Y \cup \{\alpha = \beta \rightarrow b\}$ has mgu $\mu\varphi$, where

$X = \{\delta = \varepsilon \mid$ there is a variable that is free with relative type δ in some t_j, where x_j is free in one of u_1 or u_2 and is either free with relative type ε in the other u_i or is free with relative type ε in some t_p, where x_p is free in the other $u_i\}$, and

$Y = \{\gamma_1 = \gamma_2 \mid$ there is a variable that is free with relative type γ_i in u_i, for $i = 1, 2\}$

[by Part 1 of Proposition 2.2.1, since θ is idempotent, μ is an mgu for $Y \cup \{\alpha = \beta \rightarrow b\}$, and $(U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup U_{u_2,\theta|_{u_2}} \cup V_{u_2,\theta|_{u_2}} \cup X)\mu = U_{(u_1\ u_2),\theta} \cup V_{(u_1\ u_2),\theta}$]

implies there exist type substitutions φ_1, φ_2 and η such that $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ has mgu φ_i, for $i = 1, 2$, and $(X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$ has mgu η, where $\mu\varphi = (\varphi_1 \cup \varphi_2)\eta$
[by Proposition 2.2.2]

implies there exist type substitutions φ_1, φ_2 and η such that $u_1\theta$ is a term of type $\alpha\varphi_1$, $u_2\theta$ is a term of type $\beta\varphi_2$ and, if y is a free variable in $u_i\theta$ and y has relative type δ in u_i or some t_j, then y has relative type $\delta\varphi_i$ in $u_i\theta$, for $i = 1, 2$, and $(X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$ has mgu η, where $\mu\varphi = (\varphi_1 \cup \varphi_2)\eta$
[by the induction hypothesis]

implies there exist type substitutions φ_1, φ_2 and η such that $(u_1\ u_2)\theta$ is a term of type $b\eta$; if y is a free variable in $(u_1\ u_2)\theta$ and y has relative type ϱ in some $u_i\theta$, then y has relative type $\varrho\eta$ in $(u_1\ u_2)\theta$; and, if y is a free variable in $u_i\theta$ and y has relative type δ in u_i or some t_j, then y has relative type $\delta\varphi_i$ in $u_i\theta$, for $i = 1, 2$, and $(X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$ has mgu η; where $\mu\varphi = (\varphi_1 \cup \varphi_2)\eta$
[by the definition of an application, since each x_i occurs freely exactly once in $(u_1\ u_2)$ and therefore $\mathit{Constraints}_{(u_1\ u_2)\theta} = (X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$]

implies $(u_1\ u_2)\theta$ is a term of type $\sigma\varphi$ and, if y is a free variable in $(u_1\ u_2)\theta$ and y has relative type δ in $(u_1\ u_2)$ or some t_i, then y has relative type $\delta\varphi$ in $(u_1\ u_2)\theta$.
[$(u_1\ u_2)\theta$ has type $b\eta$, where $b\eta = b(\varphi_1 \cup \varphi_2)\eta = b\mu\varphi = \sigma\varphi$.
Let y be a free variable in $(u_1\ u_2)\theta$. There are two cases.
(i) Suppose that y has relative type δ in $(u_1\ u_2)$. Hence y has relative type δ' in some u_j, where $\delta'\mu = \delta$, and so y has relative type $\delta'\varphi_j$ in $u_j\theta$. Thus y has relative type $\delta'\varphi_j\eta$ in $(u_1\ u_2)\theta$, where $\delta'\varphi_j\eta = \delta'(\varphi_1 \cup \varphi_2)\eta = \delta'\mu\varphi = \delta\varphi$.
(ii) Suppose that y has relative type δ in some t_i and x_i is free in u_j. Hence y has relative type $\delta\varphi_j$ in $u_j\theta$. Thus y has relative type $\delta\varphi_j\eta$ in $(u_1\ u_2)\theta$, where $\delta\varphi_j\eta = \delta(\varphi_1 \cup \varphi_2)\eta = \delta\mu\varphi = \delta\varphi$]

Suppose that t has the form $(u_1, \ldots, u_n)$ with type σ, where u_i has type σ_i, for $i = 1, \ldots, n$, and μ is the associated mgu for $(u_1, \ldots, u_n)$. Hence $\sigma = (\sigma_1 \times \cdots \times \sigma_n)\mu$. Then

$U_{(u_1,\ldots,u_n),\theta} \cup V_{(u_1,\ldots,u_n),\theta}$ has mgu φ

implies $U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup \cdots \cup U_{u_n,\theta|_{u_n}} \cup V_{u_n,\theta|_{u_n}} \cup X \cup Y$ has mgu $\mu\varphi$, where

$X = \{\delta_{j_1} = \cdots = \delta_{j_m} [= \varepsilon] \mid$ there is a variable that is free with relative type δ_{j_p} in some t_i, and x_i occurs freely in u_{j_p} $(1 \leq p \leq m)$ and, possibly, is free with relative type ε in some $u_j\}$, and

$Y = \{\gamma_{j_1} = \cdots = \gamma_{j_m} \mid$ there is a variable that is free with relative type γ_{j_p} in u_{j_p} $(1 \leq p \leq m$ and $m > 1)\}$

[by Part 1 of Proposition 2.2.1, since θ is idempotent, μ is an mgu for Y, and $(U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup \cdots \cup U_{u_n,\theta|_{u_n}} \cup V_{u_n,\theta|_{u_n}} \cup X)\mu = U_{(u_1,\ldots,u_n),\theta} \cup V_{(u_1,\ldots,u_n),\theta}$]

implies there exist type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ has mgu φ_i, for $i = 1, \ldots, n$, and $(X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η, where $\mu\varphi = (\varphi_1 \cup \cdots \cup \varphi_n)\eta$

[by Proposition 2.2.2]

implies there exist type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $u_i\theta$ is a term of type $\sigma_i\varphi_i$ and, if y is a free variable in $u_i\theta$ and y has relative type δ in u_i or some t_j, then y has relative type $\delta\varphi_i$ in $u_i\theta$, for $i = 1, \ldots, n$, and $(X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η, where $\mu\varphi = (\varphi_1 \cup \cdots \cup \varphi_n)\eta$

[by the induction hypothesis]

implies there exist type substitutions $\varphi_1, \ldots, \varphi_n$ and η such that $(u_1, \ldots, u_n)\theta$ is a term of type $(\sigma_1\varphi_1 \times \cdots \times \sigma_n\varphi_n)\eta$; if y is a free variable in $(u_1, \ldots, u_n)\theta$ and y has relative type ϱ in some $u_i\theta$, then y has relative type $\varrho\eta$ in $(u_1, \ldots, u_n)\theta$; and, if y is a free variable in $u_i\theta$ and y has relative type δ in u_i or some t_j, then y has relative type $\delta\varphi_i$ in $u_i\theta$, for $i = 1, \ldots, n$, and $(X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$ has mgu η; where $\mu\varphi = (\varphi_1 \cup \cdots \cup \varphi_n)\eta$

[by the definition of a tuple, since each x_i occurs freely exactly once in $(u_1, \ldots, u_n)$ and therefore $\mathit{Constraints}_{(u_1,\ldots,u_n)\theta} = (X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$]

implies $(u_1, \ldots, u_n)\theta$ is a term of type $\sigma\varphi$ and, if y is a free variable in $(u_1, \ldots, u_n)\theta$ and y has relative type δ in $(u_1, \ldots, u_n)$ or some t_i, then y has relative type $\delta\varphi$ in $(u_1, \ldots, u_n)\theta$

[$(u_1, \ldots, u_n)\theta$ has type $(\sigma_1\varphi_1 \times \cdots \times \sigma_n\varphi_n)\eta$, where $(\sigma_1\varphi_1 \times \cdots \times \sigma_n\varphi_n)\eta = (\sigma_1 \times \cdots \times \sigma_n)(\varphi_1 \cup \cdots \cup \varphi_n)\eta = (\sigma_1 \times \cdots \times \sigma_n)\mu\varphi = \sigma\varphi$.

Let y be a free variable in $(u_1, \ldots, u_n)\theta$. There are two cases.

(i) Suppose that y has relative type δ in $(u_1, \ldots, u_n)$. Hence y has relative type δ' in some u_j, where $\delta'\mu = \delta$, and so y has relative type $\delta'\varphi_j$ in $u_j\theta$. Thus y has relative type $\delta'\varphi_j\eta$ in $(u_1, \ldots, u_n)\theta$, where $\delta'\varphi_j\eta = \delta'(\varphi_1 \cup \cdots \cup \varphi_n)\eta = \delta'\mu\varphi = \delta\varphi$.

(ii) Suppose that y has relative type δ in some t_i and x_i is free in u_j. Hence y has relative type $\delta\varphi_j$ in $u_j\theta$. Thus y has relative type $\delta\varphi_j\eta$ in $(u_1, \ldots, u_n)\theta$, where $\delta\varphi_j\eta = \delta(\varphi_1 \cup \cdots \cup \varphi_n)\eta = \delta\mu\varphi = \delta\varphi$] □

The next example shows that the condition in Proposition 2.5.2 that each x_i occurs freely exactly once in t cannot be dropped.

Example 2.5.4. Let t be (x, x) with type $a \times a$, for some parameter a, and $\theta = \{x/[]\}$, where [] has type *List* b, for some parameter b. Then $t\theta = ([], [])$, which has type *List* $c \times$ *List* d, for some parameters c and d. However, the type substitution φ from Proposition 2.5.2 for this example is $\{a/\mathit{List}\ b\}$ and $(a \times a)\{a/\mathit{List}\ b\} = \mathit{List}\ b \times \mathit{List}\ b$, which is not (a variant of) $\mathit{List}\ c \times \mathit{List}\ d$.

The following result considers the situation when the condition that each x_i occurs freely exactly once in t is dropped. In this case, it is no longer true that $t\theta$ has type $\sigma\varphi$; instead its type may be more general than $\sigma\varphi$. To see

why this is the case, consider a tuple t containing several occurrences of a free variable x in distinct components of the tuple. These occurrences of x constrain the type of t, by the definition of a tuple. However, a substitution θ may contain a binding x/s for which s may contain no free variables. Thus the constraint given by the occurrences of x may be lost in $t\theta$, as in the last example. Thus $t\theta$ may have a type strictly more general that $\sigma\varphi$.

Proposition 2.5.3. *Let t be a term having type σ and $x_1, \ldots, x_n$ be free variables in t. Let $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ be an idempotent term substitution. If $U_{t,\theta} \cup V_{t,\theta}$ has mgu φ, then $t\theta$ is a term of type χ, where $\sigma\varphi = \chi\pi$, and, if y is a free variable in $t\theta$ and y has relative type δ in t or some t_i, then y has relative type ω in $t\theta$, where $\delta\varphi = \omega\pi$, for some type substitution π.*

Proof. The proof is left as an exercise. □

Proposition 2.5.2 shows that if $U_{t,\theta} \cup V_{t,\theta}$ is unifiable, then $t\theta$ is term. The next result shows that the converse of this is true.

Proposition 2.5.4. *Let t be a term and $x_1, \ldots, x_n$ variables that each occur freely exactly once in t. Let $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ be an idempotent term substitution. If $t\theta$ is a term, then $U_{t,\theta} \cup V_{t,\theta}$ is unifiable.*

Proof. The proof proceeds by induction on the structure of terms.

In case t is a variable or constant, the result is obvious.

Suppose t has the form $\lambda x.s$. Then

$(\lambda x.s)\theta$ is a term
implies $s\theta$ is a term
[It can be assumed without loss of generality that $x \notin range(\theta)$]
implies $U_{s,\theta} \cup V_{s,\theta}$ is unifiable
[by the induction hypothesis]
implies $U_{\lambda x.s,\theta} \cup V_{\lambda x.s,\theta}$ is unifiable
[since $U_{s,\theta} = U_{\lambda x.s,\theta}$ and $V_{s,\theta} = V_{\lambda x.s,\theta}$]

Suppose t has the form $(u_1\ u_2)$, where u_1 has type α and u_2 has type β. Then

$(u_1\ u_2)\theta$ is a term
implies $u_1\theta$ and $u_2\theta$ are terms and $Constraints_{(u_1\ u_2)\theta}$ is unifiable
[by Proposition 2.3.3]
implies $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ is unifiable ($i = 1, 2$) and $Constraints_{(u_1\ u_2)\theta}$ is unifiable
[by the induction hypothesis]
implies $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ has mgu φ_i, say ($i = 1, 2$) and $(X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$ is unifiable, where X and Y are defined as for the corresponding part of Proposition 2.5.2
[by Proposition 2.5.2, since $Constraints_{(u_1\ u_2)\theta} = (X \cup Y \cup \{\alpha = \beta \to b\})(\varphi_1 \cup \varphi_2)$]

implies $U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup U_{u_2,\theta|_{u_2}} \cup V_{u_2,\theta|_{u_2}} \cup X \cup Y \cup \{\alpha = \beta \to b\}$ is unifiable
implies $U_{(u_1\ u_2),\theta} \cup V_{(u_1\ u_2),\theta}$ is unifiable.
[by Part 3 of Proposition 2.2.1, since $Constraints_{(u_1\ u_2)} = Y \cup \{\alpha = \beta \to b\}$ and, if μ is an mgu of $Y \cup \{\alpha = \beta \to b\}$, then $(U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup U_{u_2,\theta|_{u_2}} \cup V_{u_2,\theta|_{u_2}} \cup X)\mu = U_{(u_1\ u_2),\theta} \cup V_{(u_1\ u_2),\theta}$]

Suppose t has the form $(u_1, \ldots, u_n)$. Then

$(u_1, \ldots, u_n)\theta$ is a term
implies $u_1\theta, \ldots, u_n\theta$ are terms and $Constraints_{(u_1,\ldots,u_n)\theta}$ is unifiable
[by Proposition 2.3.3]
implies $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ is unifiable $(i = 1, \ldots, n)$ and $Constraints_{(u_1,\ldots,u_n)\theta}$ is unifiable
[by the induction hypothesis]
implies $U_{u_i,\theta|_{u_i}} \cup V_{u_i,\theta|_{u_i}}$ has mgu φ_i, say $(i = 1, \ldots, n)$ and $(X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$ is unifiable, where X and Y are defined as for the corresponding part of Proposition 2.5.2
[by Proposition 2.5.2, since $Constraints_{(u_1,\ldots,u_n)\theta} = (X \cup Y)(\varphi_1 \cup \cdots \cup \varphi_n)$]
implies $U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup \cdots \cup U_{u_n,\theta|_{u_n}} \cup V_{u_n,\theta|_{u_n}} \cup X \cup Y$ is unifiable
implies $U_{(u_1,\ldots,u_n),\theta} \cup V_{(u_1,\ldots,u_n),\theta}$ is unifiable
[by Part 3 of Proposition 2.2.1, since $Constraints_{(u_1,\ldots,u_n)} = Y$ and, if μ is an mgu of Y, then $(U_{u_1,\theta|_{u_1}} \cup V_{u_1,\theta|_{u_1}} \cup \cdots \cup U_{u_n,\theta|_{u_n}} \cup V_{u_n,\theta|_{u_n}} \cup X)\mu = U_{(u_1,\ldots,u_n),\theta} \cup V_{(u_1,\ldots,u_n),\theta}$] □

The following result establishes a relationship between $\precsim$ and instantiation by a term substitution that will be used in Chap. 5 to show that the run-time system for a programming language based on the logic does not need to do type checking.

Proposition 2.5.5. *Let s and t be terms, where each free variable in t occurs freely exactly once in t, and θ an idempotent term substitution. If $t\theta$ is a term and $s \precsim t$, then $s\theta$ is a term and $s\theta \precsim t\theta$.*

Proof. Let s have type σ, t have type τ, $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ and $Subst_{s \precsim t} = \xi$. Since $t\theta$ is a term and t has no repeated free variables, $U_{t,\theta|_t} \cup V_{t,\theta|_t}$ is unifiable, by Proposition 2.5.4. Let φ be an mgu for $U_{t,\theta|_t} \cup V_{t,\theta|_t}$. By Proposition 2.5.2, $t\theta$ has type $\tau\varphi$ and, if y is a free variable in $t\theta$ and y has relative type δ in t or some t_i, then y has relative type $\delta\varphi$ in $t\theta$.

Since $s \precsim t$, $\xi\varphi$ is a unifier for $U_{s,\theta|_s} \cup V_{s,\theta|_s}$ and thus $U_{s,\theta|_s} \cup V_{s,\theta|_s}$ has mgu ψ, where $\xi\varphi = \psi\alpha$, for some type substitution α. By Proposition 2.5.3, $s\theta$ is a term of type χ, where $\sigma\psi = \chi\beta$, and, if y is a free variable in $s\theta$ and y has relative type δ in s of some t_i, then y has relative type ω in $s\theta$, where $\delta\psi = \omega\beta$, for some type substitution β. It remains to show that $Subst_{s\theta \precsim t\theta} = \beta\alpha$.

First, $s\theta$ has type χ, $t\theta$ has type $\tau\varphi$, and $\chi\beta\alpha = \sigma\psi\alpha = \sigma\xi\varphi = \tau\varphi$. Finally, let y be a free variable in $s\theta$. Then y is also a free variable in $t\theta$. Suppose y has

relative type δ in s or some t_i. Thus y has relative ω in $s\theta$, where $\delta\psi = \omega\beta$, y has relative type $\delta\xi\varphi$ in $t\theta$, where $\xi\varphi = \psi\alpha$, and $\omega\beta\alpha = \delta\psi\alpha = \delta\xi\varphi$. □

The following example shows that the condition in Proposition 2.5.5 that each free variable in t occur freely exactly once in t cannot be dropped.

Example 2.5.5. Let $f : a \to a \times a$ and $g : a \times b \to a \times b$ be constants. Let $s = (f\ x)$, $t = (g\ (x, x))$, and $\theta = \{x/[]\}$. Note that both s and t have type $a \times a$. Then $s \precsim t$, $s\theta = (f\ [])$ is term of type $List\ a \times List\ a$, $t\theta = (g\ ([], []))$ is a term of type $List\ a \times List\ b$, but $s\theta \not\precsim t\theta$.

This section concludes with a result about term replacement that will be needed for predicate construction. For this, a preliminary result about replacement by a variable is needed.

Proposition 2.5.6. *Let t be a term, s a subterm of t at occurrence o, and x a variable not appearing in t. Then $t[s/x]_o$ is a term. Furthermore, if t has type τ and $t[s/x]_o$ has type τ', then there exists a type substitution γ such that $\tau = \tau'\gamma$, and, if the relative type of a free variable other than x in $t[s/x]_o$ is δ, then the relative type of this free variable in t is $\delta\gamma$.*

Proof. Introduce a new constant C of type a, for some parameter a, into the alphabet. Then $C \precsim s$. Thus, $t[s/C]_o$ is a term and $t[s/C]_o \precsim t$, by Part 1 of Proposition 2.4.6. It follows from this that $t[s/x]_o$ has the desired properties. □

Definition 2.5.4. Let t be a term and s a subterm of t at occurrence o. Then the *positional type* of s in t at o is the relative type of x in $t[s/x]_o$, where x is a variable not appearing in t.

Definition 2.5.5. Let t be a term and s a subterm of t at occurrence o. Then the *residual type* of t with respect to s at o is the type of $t[s/x]_o$, where x is a variable not appearing in t.

Example 2.5.6. Let $[] : List\ a$, $\sharp : a \to List\ a \to List\ a$, $transform : List\ a \to Tree\ a$, and $A, B : M$, where M is a nullary type constructor and *Tree* is a unary type constructor. Consider the term $transform\ (A \sharp B \sharp [])$ of type $Tree\ M$ and the subterm $A \sharp B \sharp []$ at occurrence 2. Then the positional type of $A \sharp B \sharp []$ in $transform\ (A \sharp B \sharp [])$ at 2 is $List\ a$. Also the residual type of $transform\ (A \sharp B \sharp [])$ with respect to $A \sharp B \sharp []$ at 2 is $Tree\ a$.

Proposition 2.5.7. *Let t be a term of type τ, s a subterm of t at occurrence o, σ the positional type of s in t at o, r a closed term of type ϱ, and δ the residual type of t with respect to s at o. If $\sigma = \varrho$ has a most general unifier, say, ξ, then $t[s/r]_o$ is a term of type $\delta\xi$. Conversely, if $t[s/r]_o$ is a term, then $\sigma = \varrho$ is unifiable.*

Proof. The first part of the proposition follows immediately from Proposition 2.5.2. The converse follows immediately from Proposition 2.5.4. □

Proposition 2.5.8. *Let t be a term, s_i a subterm of t at occurrence o_i, and r_i a closed term, for $i = 1, \ldots, n$, such that the set of subterms $\{s_i\}_{i=1}^n$ is disjoint. Suppose that the type of s_i is more general than the type of r_i, for $i = 1, \ldots, n$, and $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$ is a term. Then the type of t is more general than the type of $t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$.*

Proof. The proof is by induction on n. First the result is proved for $n = 1$. Thus suppose s is a subterm at occurrence o, r a closed subterm, the type of s is σ, the positional type of s in t is α, the residual type of t with respect to s is β, the type of r is ϱ, and $t[s/r]_o$ is a term. It has to be shown that the type of t is more general than the type of $t[s/r]_o$.

Let t' be the term $t[s/x]_o$, where x is a variable not appearing in t. Then t is $t'[x/s]_o$ and $t[s/r]_o$ is $t'[x/r]_o$. It suffices to show that the type of $t'[x/s]_o$ is more general than the type of $t'[x/r]_o$. Since σ is more general than ϱ, there exists a type substitution ξ such that $\sigma\xi = \varrho$. According to Proposition 2.5.2, the type of $t'[x/s]_o$ is $\beta\psi$, where ψ is an mgu of $\alpha = \sigma$. Similarly, the type of $t'[x/s]_o$ is $\beta\varphi$, where φ is an mgu of $\alpha = \varrho$. Now $\alpha\xi\varphi = \alpha\varphi = \varrho\varphi = \sigma\xi\varphi$, because ξ can be assumed to not act on α. Since ψ is an mgu of $\alpha = \sigma$, it follows that $\xi\varphi = \psi\eta$, for some η. Thus $\beta\varphi = \beta\xi\varphi = \beta\psi\eta$, since ξ can be assumed to not act on β. Hence the type of $t'[x/s]_o$ is more general than the type of $t'[x/r]_o$.

Assume now that the result holds for n. Consider the term $t[s_1/r_1, \ldots, s_{n+1}/r_{n+1}]_{o_1,\ldots,o_{n+1}}$, which is by definition equal to $(t[s_1/r_1]_{o_1})[s_2/r_2, \ldots, s_{n+1}/r_{n+1}]_{o_2,\ldots,o_{n+1}}$. By the argument from the base case, the type of t is more general than the type of $t[s_1/r_1]_{o_1}$ and, by the induction hypothesis, the type of $t[s_1/r_1]_{o_1}$ is more general than the type of $(t[s_1/r_1]_{o_1})[s_2/r_2, \ldots, s_{n+1}/r_{n+1}]_{o_2,\ldots,o_{n+1}}$. Hence the type of t is more general than the type of $t[s_1/r_1, \ldots, s_{n+1}/r_{n+1}]_{o_1,\ldots,o_{n+1}}$. □

2.6 λ-Conversion

The results of this section are concerned with rules of λ-conversion, which have to be handled carefully because of the polymorphic nature of the logic. The main result is that if $(\lambda x.s\ t)$ is a term, then $s\{x/t\}$ is a term and $s\{x/t\} \precsim (\lambda x.s\ t)$. This result provides the basis for β-reduction.

Proposition 2.6.1. *If $(\lambda x.(s_1\ s_2)\ t)$ is a term, then $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$ is a term and $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t)) \precsim (\lambda x.(s_1\ s_2)\ t)$.*

Proof. Suppose that $(\lambda x.(s_1\ s_2)\ t)$ is a term, where the type of s_1 is σ_1, s_2 is σ_2, and t is τ. Let φ be an mgu for $Constraints_{(s_1\ s_2)}$, where $Constraints_{(s_1\ s_2)}$ is

$$\{\sigma_1 = \sigma_2 \to b\} \cup$$
$$\{\delta_1 = \delta_2 \mid \text{there is a variable that is free with relative type } \delta_i \text{ in } s_i, \; i = 1, 2\}.$$

(It can be assumed without loss of generality that φ acts only on the parameters in $Constraints_{(s_1\ s_2)}$.) Now $Constraints_{(\lambda x.(s_1\ s_2)\ t)}$ is

$$\{\varrho = \tau\} \cup$$
$$\{\delta = \varepsilon \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta \text{ in } (s_1\ s_2) \text{ and free with relative type } \varepsilon \text{ in } t\},$$

where ϱ is the relative type of x in $(s_1\ s_2)$, if x is free in $(s_1\ s_2)$; otherwise, ϱ is a new parameter. Suppose that $Constraints_{(\lambda x.(s_1\ s_2)\ t)}$ has mgu η. Hence, by Part 1 of Proposition 2.2.1, $\varphi\eta$ is an mgu of

$$\{\sigma_1 = \sigma_2 \to b\} \cup$$
$$\{\delta_1 = \delta_2 \mid \text{there is a variable that is free with relative type } \delta_i \text{ in } s_i, \; i = 1, 2\} \cup$$
$$\{[\varrho_1 =][\varrho_2 =]\tau\} \cup$$
$$\{[\delta_1 =][\delta_2 =]\varepsilon \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_i \text{ in } s_i, \text{where } i = 1 \text{ or } 2 \text{ or both, and is also free with relative type } \varepsilon \text{ in } t\},$$

where ϱ_i is the relative type of x in s_i, $i = 1, 2$. If x is not free in $(s_1\ s_2)$, then the third equation becomes $a = \tau$, for some new parameter a.

Suppose that the parameters in τ and the relative types of variables in t are standardised apart in two distinct ways. To distinguish the two renamings, I denote by $t^{(i)}$ the term t, where its parameters are understood to be renamed in the ith way, by $\tau^{(i)}$ the corresponding renaming of τ, and by $\varepsilon^{(i)}$ the relative type of a free variable in $t^{(i)}$, where ε is the relative type of the variable in t. Let ξ be the type substitution that maps each set of the renamed parameters back onto the original set of parameters. Thus $\tau^{(i)}\xi = \tau$ and $\varepsilon^{(i)}\xi = \varepsilon$, for $i = 1, 2$. Now $\xi\varphi\eta$ is a unifier of the set X of equations, where X is

$$\{\sigma_1 = \sigma_2 \to b\} \cup$$
$$\{\omega_1 = \omega_2 \mid \text{there is a variable other than } x \text{ that is free with relative type } \omega_i \text{ in } s_i \text{ or } t^{(i)}, i = 1, 2\} \cup$$
$$\{\varrho_i = \tau^{(i)} \mid i = 1, 2\} \cup$$
$$\{\delta_i = \varepsilon^{(i)} \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_i \text{ in } s_i \text{ and free with relative type } \varepsilon^{(i)} \text{ in } t^{(i)}, i = 1, 2\},$$

where ϱ_i is the relative type of x in s_i, if x is free in s_i; otherwise, ϱ_i is a new parameter, $i = 1, 2$. Let π be an mgu for X. Thus $\xi\varphi\eta = \pi\beta$, for some type substitution β.

Now $\mathit{Constraints}_{(\lambda x.s_i\ t)}$ is

$$\{\varrho_i = \tau^{(i)}\} \cup$$
$$\{\delta_i = \varepsilon^{(i)} \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_i \text{ in } s_i \text{ and free with relative type } \varepsilon^{(i)} \text{ in } t^{(i)}\}.$$

By Proposition 2.2.2, there exists type substitutions η_1, η_2 and μ such that η_i is an mgu for $\mathit{Constraints}_{(\lambda x.s_i\ t)}$, the parameters in η_i all appear in $\mathit{Constraints}_{(\lambda x.s_i\ t)}$ $(i = 1, 2)$, μ is an mgu for $Y(\eta_1 \cup \eta_2)$, and $\pi = (\eta_1 \cup \eta_2)\mu$, where Y is

$$\{\sigma_1 = \sigma_2 \to b\} \cup$$
$$\{\omega_1 = \omega_2 \mid \text{there is a variable other than } x \text{ that is free with relative type } \omega_i \text{ in } s_i \text{ or } t^{(i)}, i = 1, 2\}.$$

Thus $(\lambda x.s_i\ t)$ is a term, for $i = 1, 2$. Furthermore, $\mathit{Constraints}_{((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))}$ $= Y(\eta_1 \cup \eta_2)$, and thus $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$ is a term.

The proof concludes by showing that $\mathit{Subst}_{((\lambda x.s_1\ t)\ (\lambda x.s_2\ t)) \precsim (\lambda x.(s_1\ s_2)\ t)} =$ β. First, note that $(\lambda x.(s_1\ s_2)\ t)$ has type $b\varphi\eta$, $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$ has type $b\mu$, and $b\varphi\eta = b\xi\varphi\eta = b\pi\beta = b(\eta_1 \cup \eta_2)\mu\beta = b\mu\beta$.

Let y be a free variable in $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$. Hence y is also free in $(\lambda x.(s_1\ s_2)\ t)$. Suppose that the relative type of y in $(\lambda x.(s_1\ s_2)\ t)$ is δ. There are two (not necessarily disjoint) cases.

(a) y is a free variable in $\lambda x.(s_1\ s_2)$. Thus y is free in some s_i and $\delta = \delta_i\varphi\eta$, where δ_i is the relative type of y in s_i. Also y has relative type $\delta_i\eta_i\mu$ in $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$. But $\delta = \delta_i\varphi\eta = \delta_i\xi\varphi\eta = \delta_i\pi\beta = \delta_i(\eta_1 \cup \eta_2)\mu\beta = \delta_i\eta_i\mu\beta$.

(b) y is a free variable in t. Suppose that the relative type of y in t is δ'. Thus $\delta = \delta'\eta$. Let δ'' be the relative type of y in $t^{(1)}$. Thus $\delta' = \delta''\xi$ and the relative type of y in $((\lambda x.s_1\ t)\ (\lambda x.s_2\ t))$ is $\delta''\eta_1\mu$. But $\delta = \delta'\eta = \delta'\varphi\eta =$ $\delta''\xi\varphi\eta = \delta''\pi\beta = \delta''(\eta_1 \cup \eta_2)\mu\beta = \delta''\eta_1\mu\beta$. □

The type of $((\lambda x.u\ t)\ (\lambda x.v\ t))$ may be strictly weaker than the type of $(\lambda x.(u\ v)\ t)$.

Example 2.6.1. Let f have signature $\mathit{List}\ a \to \mathit{List}\ b \to \mathit{List}\ a \times \mathit{List}\ b$, where a and b are parameters. Then $(\lambda x.((f\ x)\ x)\ [])$ is a term of type $\mathit{List}\ a \times \mathit{List}\ a$. Now $((\lambda x.(f\ x)\ [])\ (\lambda x.x\ []))$ is a term that has type $\mathit{List}\ a \times \mathit{List}\ b$ which is strictly weaker than $\mathit{List}\ a \times \mathit{List}\ a$.

Proposition 2.6.2. *If* $(\lambda x.(s_1, \ldots, s_n)\ t)$ *is a term, then* $((\lambda x.s_1\ t), \ldots,$ $(\lambda x.s_n\ t))$ *is a term and* $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \precsim (\lambda x.(s_1, \ldots, s_n)\ t)$.

Proof. Suppose that $(\lambda x.(s_1, \ldots, s_n)\ t)$ is a term, where the type of s_i is σ_i, for $i = 1, \ldots, n$, and the type of t is τ. Let φ be an mgu for $Constraints_{(s_1,\ldots,s_n)}$, where $Constraints_{(s_1,\ldots,s_n)}$ is

$$\{\delta_{i_1} = \cdots = \delta_{i_k} \mid \text{there is a variable that is free with relative type } \delta_{i_j} \text{ in } s_{i_j}, j = 1, \ldots, k \text{ and } k > 1\}.$$

(It can be assumed without loss of generality that φ acts only on the parameters in $Constraints_{(\lambda x.(s_1,\ldots,s_n)\ t)}$.) Now $Constraints_{(\lambda x.(s_1,\ldots,s_n)\ t)}$ is

$$\{\varrho = \tau\} \cup$$
$$\{\delta = \varepsilon \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta \text{ in } (s_1, \ldots, s_n) \text{ and free with relative type } \varepsilon \text{ in } t\},$$

where ϱ is the relative type of x in $(s_1, \ldots, s_n)$, if x is free in $(s_1, \ldots, s_n)$; otherwise, ϱ is a new parameter. Suppose that $Constraints_{(\lambda x.(s_1,\ldots,s_n)\ t)}$ has mgu η. Hence, by Part 1 of Proposition 2.2.1, $\varphi\eta$ is an mgu of

$$\{\delta_{i_1} = \cdots = \delta_{i_k} \mid \text{there is a variable that is free with relative type } \delta_{i_j} \text{ in } s_{i_j}, j = 1, \ldots, k \text{ and } k > 1\} \cup$$
$$\{\varrho_{i_1} = \cdots = \varrho_{i_k} = \tau\} \cup$$
$$\{\delta_{i_1} = \cdots = \delta_{i_k} = \varepsilon \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_{i_j} \text{ in } s_{i_j}, j = 1, \ldots, k \text{ and } k \geq 1, \text{ and is also free with relative type } \varepsilon \text{ in } t\},$$

where ϱ_{i_j} is the relative type of x in s_{i_j}, $j = 1, \ldots, k$ and $k \geq 0$. If $k = 0$, this equation is $a = \tau$, where a is a new parameter.

Suppose that the parameters in τ and the relative types of variables in t are standardised apart in n distinct ways. To distinguish the various renamings, I denote by $t^{(i)}$ the term t, where its parameters are understood to be renamed in the ith way, by $\tau^{(i)}$ the corresponding renaming of τ, and by $\varepsilon^{(i)}$ the relative type of a free variable in $t^{(i)}$, where ε is the relative type of the variable in t. Let ξ be the type substitution that maps each set of the renamed parameters back onto the original set of parameters. Thus $\tau^{(i)}\xi = \tau$ and $\varepsilon^{(i)}\xi = \varepsilon$, for $i = 1, \ldots, n$. Now $\xi\varphi\eta$ is a unifier of the set X of equations, where X is

$$\{\omega_{i_1} = \cdots = \omega_{i_k} \mid \text{there is a variable other than } x \text{ that is free with relative type } \omega_{i_j} \text{ in } s_{i_j} \text{ or } t^{(i_j)}, j = 1, \ldots, k \text{ and } k > 1\} \cup$$
$$\{\varrho_i = \tau^{(i)} \mid i = 1, \ldots, n\} \cup$$
$$\{\delta_i = \varepsilon^{(i)} \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_i \text{ in } s_i \text{ and free with relative type } \varepsilon^{(i)} \text{ in } t^{(i)}, i = 1, \ldots, n\},$$

where ϱ_i is the relative type of x in s_i, if x is free in s_i; otherwise, ϱ_i is a new parameter, $i = 1, \ldots, n$. Let π be an mgu for X. Thus $\xi\varphi\eta = \pi\beta$, for some type substitution β.

Now $\mathit{Constraints}_{(\lambda x.s_i\ t)}$ is

$$\{\varrho_i = \tau^{(i)}\} \cup$$
$$\{\delta_i = \varepsilon^{(i)} \mid \text{there is a variable other than } x \text{ that is free with relative type } \delta_i \text{ in } s_i \text{ and free with relative type } \varepsilon^{(i)} \text{ in } t^{(i)}\}.$$

By Proposition 2.2.2, there exists type substitutions $\eta_1, \ldots, \eta_n$ and μ such that η_i is an mgu for $\mathit{Constraints}_{(\lambda x.s_i\ t)}$, the parameters in η_i all appear in $\mathit{Constraints}_{(\lambda x.s_i\ t)}$ $(i = 1, \ldots, n)$, μ is an mgu for $Y(\eta_1 \cup \cdots \cup \eta_n)$, and $\pi = (\eta_1 \cup \cdots \cup \eta_n)\mu$, where Y is

$$\{\omega_{i_1} = \cdots = \omega_{i_k} \mid \text{there is a variable other than } x \text{ that is free with relative type } \omega_{i_j} \text{ in } s_{i_j} \text{ or } t^{(i_j)}, j = 1, \ldots, k \text{ and } k > 1\}.$$

Thus $(\lambda x.s_i\ t)$ is a term, for $i = 1, \ldots, n$. Also $\mathit{Constraints}_{((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))}$ $= Y(\eta_1 \cup \cdots \cup \eta_n)$, and thus $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$ is a term.

The last step is to show that $\mathit{Subst}_{((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \precsim (\lambda x.(s_1, \ldots, s_n)\ t)} = \beta$. First, note that $(\lambda x.(s_1, \ldots, s_n)\ t)$ has type $(\sigma_1 \times \cdots \times \sigma_n)\varphi\eta$, $((\lambda x.s_1\ t), \ldots,$ $(\lambda x.s_n\ t))$ has type $(\sigma_1\eta_1 \times \cdots \times \sigma_n\eta_n)\mu$, and

$$\begin{aligned}(\sigma_1 \times \cdots \times \sigma_n)\varphi\eta &= (\sigma_1 \times \cdots \times \sigma_n)\xi\varphi\eta \\ &= (\sigma_1 \times \cdots \times \sigma_n)\pi\beta \\ &= (\sigma_1 \times \cdots \times \sigma_n)(\eta_1 \cup \cdots \cup \eta_n)\mu\beta \\ &= (\sigma_1\eta_1 \times \cdots \times \sigma_n\eta_n)\mu\beta.\end{aligned}$$

Let y be a free variable in $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$. Hence y is also free in $(\lambda x.(s_1, \ldots, s_n)\ t)$. Suppose that the relative type of y in $(\lambda x.(s_1, \ldots, s_n)\ t)$ is δ. There are two (not necessarily disjoint) cases.

(a) y is a free variable in $\lambda x.(s_1, \ldots, s_n)$. Thus y is free in some s_i and $\delta = \delta_i\varphi\eta$, where δ_i is the relative type of y in s_i. Also y has relative type $\delta_i\eta_i\mu$ in $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$. But $\delta = \delta_i\varphi\eta = \delta_i\xi\varphi\eta = \delta_i\pi\beta = \delta_i(\eta_1 \cup \cdots \cup \eta_n)\mu\beta = \delta_i\eta_i\mu\beta$.

(b) y is a free variable in t. Suppose that the relative type of y in t is δ'. Thus $\delta = \delta'\eta$. Let δ'' be the relative type of y in $t^{(1)}$. Thus $\delta' = \delta''\xi$ and the relative type of y in $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$ is $\delta''\eta_1\mu$. But $\delta = \delta'\eta = \delta'\varphi\eta = \delta''\xi\varphi\eta = \delta''\pi\beta = \delta''(\eta_1 \cup \cdots \cup \eta_n)\mu\beta = \delta''\eta_1\mu\beta$. □

The type of $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$ may be strictly weaker than the type of $(\lambda x.(s_1, \ldots, s_n)\ t)$.

Example 2.6.2. Consider the term $(\lambda x.(x, x)\ [])$ with type $\mathit{List}\ a \times \mathit{List}\ a$. Then $((\lambda x.x\ []), (\lambda x.x\ []))$ is a term that has type $\mathit{List}\ a \times \mathit{List}\ b$.

Proposition 2.6.3. *If $(\lambda x.(\lambda y.r)\ t)$ is a term and y is not free in t, then $\lambda y.(\lambda x.r\ t)$ is a term and $\lambda y.(\lambda x.r\ t) \approx (\lambda x.(\lambda y.r)\ t)$.*

Proof. First, note that $Constraints_{(\lambda x.(\lambda y.r)\ t)} = Constraints_{(\lambda x.r\ t)}$, and also $\lambda y.(\lambda x.r\ t)$ and $(\lambda x.(\lambda y.r)\ t)$ have the same set of free variables, since t does not contain y as a free variable. Let ξ be an mgu of $Constraints_{(\lambda x.(\lambda y.r)\ t)}$. Suppose that r has type ϱ, x has relative type δ in r, and y has relative type ε in r. Then $\lambda y.r$ has type $\varepsilon \to \varrho$, $\lambda x.(\lambda y.r)$ has type $\delta \to (\varepsilon \to \varrho)$, and $(\lambda x.(\lambda y.r)\ t)$ has type $(\varepsilon \to \varrho)\xi$. Furthermore, if z is a free variable in $(\lambda x.(\lambda y.r)\ t)$ of relative type η in r or t, then z has relative type $\eta\xi$ in $(\lambda x.(\lambda y.r)\ t)$.

Since $Constraints_{(\lambda x.r\ t)}$ has mgu ξ, $(\lambda x.r\ t)$ is a term of type $\varrho\xi$. Thus $\lambda y.(\lambda x.r\ t)$ is a term of type $(\varepsilon \to \varrho)\xi$. Furthermore, if z is a free variable in $\lambda y.(\lambda x.r\ t)$ of relative type η in r or t, then z has relative type $\eta\xi$ in $\lambda y.(\lambda x.r\ t)$. Thus $\lambda y.(\lambda x.r\ t) \approx (\lambda x.(\lambda y.r)\ t)$. □

The condition that y is not free in t in Proposition 2.6.3 cannot be dropped.

Example 2.6.3. Let M and N be unary type constructors, and $f : M \to M$ and $g : N \to N$ be functions. Put $r = (f\ y)$ and $t = (g\ y)$. Then $(\lambda x.(\lambda y.r)\ t) = (\lambda x.(\lambda y.(f\ y))\ (g\ y))$ is a term of type $M \to M$. But $(\lambda x.r\ t) = (\lambda x.(f\ y)\ (g\ y))$ is not a term. Thus $\lambda y.(\lambda x.r\ t)$ is not a term.

Proposition 2.6.4.

1. *If y is a variable that is not free in a term t and $\lambda x.t$ is a term, then $\lambda y.(t\{x/y\})$ is a term that is type-equivalent to $\lambda x.t$.*
2. *If $(\lambda x.s\ t)$ is a term, then $s\{x/t\}$ is a term that is type-weaker than $(\lambda x.s\ t)$.*
3. *If x is a variable that is not free in a term t and $\lambda x.(t\ x)$ is a term, then t is type-equivalent to $\lambda x.(t\ x)$.*

Proof. 1. This part is obvious.

2. Suppose that $(\lambda x.s\ t)$ is a term. It is now shown by induction on the structure of s that $s\{x/t\}$ is a term that is type-weaker than $(\lambda x.s\ t)$.

Suppose that s is a variable, say, y. If $y = x$, then $s\{x/t\} = t$. Thus $s\{x/t\}$ is a term and clearly $t \approx (\lambda x.x\ t)$. If $y \neq x$, then $s\{x/t\} = y$. Thus $s\{x/t\}$ is a term and clearly $y \precsim (\lambda x.y\ t)$.

Suppose that s is a constant, say, C. Hence $s\{x/t\} = C$. Thus $s\{x/t\}$ is a term and clearly $C \precsim (\lambda x.C\ t)$.

Suppose that s is an abstraction, say, $\lambda y.r$. There are four cases to consider.

(a) If $y = x$, then $s\{x/t\} = (\lambda x.r)\{x/t\} = \lambda x.r$, and so $s\{x/t\}$ is a term. Furthermore, $\lambda x.r \precsim (\lambda x.(\lambda x.r)\ t)$. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

(b) If $y \neq x$ and y is not free in t, then $s\{x/t\} = (\lambda y.r)\{x/t\} = \lambda y.(r\{x/t\})$. By Proposition 2.6.3, $\lambda y.(\lambda x.r\ t)$ is a term and hence so is

$(\lambda x.r\ t)$. Thus, by the induction hypothesis, $r\{x/t\}$ is a term and $r\{x/t\} \precsim (\lambda x.r\ t)$. Thus $s\{x/t\}$ is a term and $\lambda y.(r\{x/t\}) \precsim \lambda y.(\lambda x.r\ t)$, by Part 3 of Proposition 2.4.5. Now $\lambda y.(\lambda x.r\ t) \approx (\lambda x.(\lambda y.r)\ t)$, by Proposition 2.6.3. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

(c) If $y \neq x$ and x is not free in r, then $s\{x/t\} = (\lambda y.r)\{x/t\} = \lambda y.r$, by Proposition 2.5.1. Thus $s\{x/t\}$ is a term. Furthermore, $\lambda y.r \precsim (\lambda x.(\lambda y.r)\ t)$, as x is not free in r and thus each variable free in $\lambda y.r$ is also free in $(\lambda x.(\lambda y.r)\ t)$. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

(d) If $y \neq x$, y is free in t and x is free in r, then $s\{x/t\} = (\lambda y.r)\{x/t\} = \lambda z.(r\{y/z\}\{x/t\})$, where z is a new variable. Let $r' = r\{y/z\}$. Then r' is a term of the same type as r, $(\lambda x.(\lambda z.r')\ t)$ is a term that is type equivalent to $(\lambda x.(\lambda y.r)\ t)$, z is not free in t, and $z \neq x$. By Part (b) above, $(\lambda z.r')\{x/t\}$ is a term and $(\lambda z.r')\{x/t\} \precsim (\lambda x.(\lambda z.r')\ t)$. Thus $s\{x/t\}$ is a term. Also $s\{x/t\} = \lambda z.(r\{y/z\}\{x/t\}) = (\lambda z.r')\{x/t\} \precsim (\lambda x.(\lambda z.r')\ t) \approx (\lambda x.(\lambda y.r)\ t)$. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

Suppose that s is an application, say, $(u\ v)$. Hence $s\{x/t\} = (u\ v)\{x/t\} = (u\{x/t\}\ v\{x/t\})$. By Proposition 2.6.1, $((\lambda x.u\ t)\ (\lambda x.v\ t))$ is a term such that $((\lambda x.u\ t)\ (\lambda x.v\ t)) \precsim (\lambda x.(u\ v)\ t)$. Thus $(\lambda x.u\ t)$ and $(\lambda x.v\ t)$ are terms. By the induction hypothesis, $u\{x/t\}$ and $v\{x/t\}$ are terms, and $u\{x/t\} \precsim (\lambda x.u\ t)$ and $v\{x/t\} \precsim (\lambda x.v\ t)$. Thus $(u\ v)\{x/t\} \precsim ((\lambda x.u\ t)\ (\lambda x.v\ t)) \precsim (\lambda x.(u\ v)\ t)$, by Part 4 of Proposition 2.4.5. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

Suppose that s is a tuple, say, $(s_1, \ldots, s_n)$. Hence $s\{x/t\} = (s_1, \ldots, s_n)\{x/t\} = (s_1\{x/t\}, \ldots, s_n\{x/t\})$. By Proposition 2.6.2, it follows that $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t))$ is a term such that $((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \precsim (\lambda x.(s_1, \ldots, s_n)\ t)$. Thus $(\lambda x.s_i\ t)$ is a term, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i\{x/t\}$ is a term and $s_i\{x/t\} \precsim (\lambda x.s_i\ t)$, for $i = 1, \ldots, n$. Thus $(s_1, \ldots, s_n)\{x/t\} \precsim ((\lambda x.s_1\ t), \ldots, (\lambda x.s_n\ t)) \precsim (\lambda x.(s_1, \ldots, s_n)\ t)$, by Part 5 of Proposition 2.4.5. That is, $s\{x/t\} \precsim (\lambda x.s\ t)$.

This completes the induction argument.

3. Suppose that t has type $\alpha \to \beta$. Then $(t\ x)$ has type β, since x is not free in t. Also x has relative type α in $(t\ x)$. Hence $\lambda x.(t\ x)$ has type $\alpha \to \beta$. Clearly t and $\lambda x.(t\ x)$ have the same set of free variables and each such free variable has the same relative type in t as it has in $\lambda x.(t\ x)$. □

It is not generally true that $s\{x/t\} \approx (\lambda x.s\ t)$. Here are two examples to illustrate what goes wrong.

Example 2.6.4. Let M be a nullary type constructor and $C : M$ a constant. Let s be C and let t be the variable y. Then $s\{x/t\}$ is C, but $(\lambda x.s\ t)$ is $(\lambda x.C\ y)$ which contains the additional free variable y. Thus $s\{x/t\} \precsim (\lambda x.s\ t)$, but $s\{x/t\} \not\approx (\lambda x.s\ t)$.

Example 2.6.5. Let $s = (x, x)$ and $t = []$. Then $(\lambda x.s\ t) = (\lambda x.(x, x)\ [])$ is a term of type $List\ a \times List\ a$. However, $s\{x/t\} = (x, x)\{x/[]\} = ([], [])$ is a term of type $List\ a \times List\ b$. Thus $s\{x/t\} \precsim (\lambda x.s\ t)$, but $s\{x/t\} \not\approx (\lambda x.s\ t)$.

Proposition 2.6.5. *Let u be a term and v a subterm of u at occurrence o.*

1. *If v is $\lambda x.t$ and y is a variable that is not free in t, then $u[\lambda x.t/\lambda y.(t\{x/y\})]_o$ is a term that is type-equivalent to u.*
2. *If v is $(\lambda x.s\ t)$, then $u[(\lambda x.s\ t)/s\{x/t\}]_o$ is a term that is type-weaker than u.*
3. *If v is $\lambda x.(t\ x)$, where x is a variable that is not free in t, then $u[\lambda x.(t\ x)/t]_o$ is a term that is type-equivalent to u.*

Proof. Parts 1 and 3 follow immediately from Propositions 2.4.7 and 2.6.4, and Part 2 from Propositions 2.4.6 and 2.6.4. □

Note that it is not generally true that $u[(\lambda x.s\ t)/s\{x/t\}]_o \approx u$.

Example 2.6.6. Let M and N be nullary type constructors, and $C : M$ and $F : M \to N$ be constants. Let s be C, t be $(F\ y)$, where y is a variable, and u be $(y, (\lambda x.C\ (F\ y)))$, which has type $M \times M$. But $u[(\lambda x.s\ t)/s\{x/t\}]_o$ is (y, C), which has type $a \times M$, for some parameter a.

I now define three relations $\succ_\alpha$, $\succ_\beta$, and $\succ_\eta$, corresponding to α-conversion, β-reduction, and η-reduction, respectively.

Definition 2.6.1. The rules of λ-conversion are as follows.

1. (α-conversion) $\lambda x.t \succ_\alpha \lambda y.(t\{x/y\})$, if y is not free in t.
2. (β-reduction) $(\lambda x.s\ t) \succ_\beta s\{x/t\}$.
3. (η-reduction) $\lambda x.(t\ x) \succ_\eta t$, if x is not free in t.

Definition 2.6.2. The relation $\longrightarrow_\alpha$ is defined by $u \longrightarrow_\alpha u[s/t]_o$ if s is a subterm of u at occurrence o and $s \succ_\alpha t$. Similarly, define $\longrightarrow_\beta$ and $\longrightarrow_\eta$. Let $\longrightarrow_{\beta\eta}$ be $\longrightarrow_\beta \cup \longrightarrow_\eta$.

Let $\stackrel{*}{\longrightarrow}_{\beta\eta}$ be the reflexive, transitive closure of $\longrightarrow_{\beta\eta}$. Similarly, define $\stackrel{*}{\longrightarrow}_\alpha$, $\stackrel{*}{\longrightarrow}_\beta$ and $\stackrel{*}{\longrightarrow}_\eta$.

Let $\stackrel{*}{\longleftrightarrow}_{\beta\eta}$ be the reflexive, symmetric, and transitive closure of $\longrightarrow_{\beta\eta}$. Similarly, define $\stackrel{*}{\longleftrightarrow}_\alpha$, $\stackrel{*}{\longleftrightarrow}_\beta$ and $\stackrel{*}{\longleftrightarrow}_\eta$.

Definition 2.6.3. If $s \stackrel{*}{\longleftrightarrow}_\alpha t$ (resp., $s \stackrel{*}{\longleftrightarrow}_\beta t$, $s \stackrel{*}{\longleftrightarrow}_{\beta\eta} t$), then s and t are said to be *α-equivalent* (resp., *β-equivalent*, *$\beta\eta$-equivalent*).

Note that α-equivalent terms differ only in the names of their bound variables.

Proposition 2.6.6. *Let s and t be terms.*

1. *If $s \stackrel{*}{\longrightarrow}_\alpha t$, then $s \approx t$.*
2. *If $s \stackrel{*}{\longrightarrow}_\beta t$, then $s \precsim t$.*
3. *If $s \stackrel{*}{\longrightarrow}_\eta t$, then $s \approx t$.*
4. *If $s \stackrel{*}{\longrightarrow}_{\beta\eta} t$, then $s \precsim t$.*

Proof. These results follow immediately from Proposition 2.6.5. □

2.7 Model Theory

Next I turn to the semantics of the logic, which is derived from the semantics for type theory originally given by Henkin. The main concept is that of an interpretation which provides a meaning for the symbols used to model an application. The development here is fairly standard, except for the complications caused by polymorphism. The modest aim is to provide the definition of an appropriate class of interpretations and, in the next section, prove a soundeness theorem (Proposition 2.8.3). The much more difficult topic of completeness is ignored since it is not relevant for the applications to learning considered here.

Definition 2.7.1. A *frame* for an alphabet is a collection $\{\mathcal{D}_\alpha\}_{\alpha \in \mathfrak{S}^c}$ of non-empty sets satisfying following conditions.

1. If α has the form $T\ \alpha_1 \ldots \alpha_k$, then $\mathcal{D}_\alpha = \{C\ d_1 \ldots d_n \mid C$ is a data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, $a_1\gamma = \alpha_1, \ldots, a_k\gamma = \alpha_k$, for some closed type substitution γ, and $d_i \in \mathcal{D}_{\sigma_i\gamma}$, for $i = 1, \ldots, n\}$.
2. If α has the form $\beta \to \gamma$, then $\mathcal{D}_\alpha$ is a collection of mappings from $\mathcal{D}_\beta$ to $\mathcal{D}_\gamma$.
3. If α has the form $\alpha_1 \times \cdots \times \alpha_n$, for some $n \geq 0$, then $\mathcal{D}_\alpha$ is the cartesian product $\mathcal{D}_{\alpha_1} \times \cdots \times \mathcal{D}_{\alpha_n}$. In particular, $\mathcal{D}_{\mathit{1}}$ is the distinguished singleton set.

For each $\alpha \in \mathfrak{S}^c$, $\mathcal{D}_\alpha$ is called a *domain.*

Definition 2.7.1 includes the construction of Herbrand universes that allow the free interpretation of the data constructors (see below). For some (closed) types, the conditions in this definition uniquely specify the corresponding domain. However, for recursive data types, such as lists, they do not.

Example 2.7.1. For Ω, the only relevant data constructors are $\top$ and $\bot$. Consequently, $\mathcal{D}_\Omega = \{\top, \bot\}$.

Example 2.7.2. For *Int*, the only relevant data constructors are the integers. Thus $\mathcal{D}_{Int} = \mathbb{Z}$.

Example 2.7.3. For *Float*, the only relevant data constructors are the floating-point numbers. Let $\mathbb{F}$ be the set of floating-point numbers. Then $\mathcal{D}_{Float} = \mathbb{F}$.

Example 2.7.4. Consider an application involving geometrical shapes, say, circles and rectangles. To model this, one could introduce a type *Shape* and two data constructors

$$Circle : Float \to Shape$$
$$Rectangle : Float \to Float \to Shape.$$

So, for example, a rectangle with length 23.5 and breadth 12.6 would be represented by the term (*Rectangle* 23.5 12.6). Then $\mathcal{D}_{Shape} = \{Circle\ f \mid f \in \mathbb{F}\} \cup \{Rectangle\ f\ g \mid f, g \in \mathbb{F}\}$.

Example 2.7.5. There are two possible domains corresponding to *List Int*. The smaller one is

$$\mathcal{D}_{List\ Int} = \{p_1 \sharp p_2 \sharp \cdots \sharp p_n \sharp [] \mid p_i \in \mathbb{Z},\ 1 \leq i \leq n, \text{ and } n \in \mathbb{N}\},$$

that is, the set of finite lists of integers.

The other domain is the set of finite or infinite lists of integers

$$\mathcal{D}'_{List\ Int} = \mathcal{D}_{List\ Int} \cup \{p_1 \sharp p_2 \sharp \cdots \mid p_i \in \mathbb{Z} \text{ and } i \geq 1\}.$$

Example 2.7.6. Assume the alphabet contains just the nullary type constructors M and N (in addition to *1* and Ω) and the data constructors $F : M \rightarrow N$ and $G : N \rightarrow M$. Then $\mathcal{D}_M = \{F\ G\ F \ldots\}$ and $\mathcal{D}_N = \{G\ F\ G \ldots\}$.

The following definition of interpretation is standard except for the introduction of closed type substitutions. Since a constant can be polymorphic, closed type substitutions are needed to give a meaning for each possible (closed) instantiation of the parameters in the signature of the constant.

Definition 2.7.2. An *interpretation* for an alphabet consists of a pair $\langle\{\mathcal{D}_\delta\}_{\delta \in \mathfrak{S}^c},\ V\rangle$, where $\{\mathcal{D}_\delta\}_{\delta \in \mathfrak{S}^c}$ is a frame for the alphabet and V is a mapping that maps each pair consisting of a constant having signature α and a closed type substitution η whose domain is the set of parameters in α to an element of $\mathcal{D}_{\alpha\eta}$ (called the *denotation* of the constant with respect to η and the interpretation), such that the following conditions are satisfied.

1. If C is a data constructor having signature $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow (T\ a_1 \ldots a_k)$ and η is $\{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$, then $V(C, \eta)$ is the element of $\mathcal{D}_{\sigma_1\eta \rightarrow \cdots \rightarrow \sigma_n\eta \rightarrow (T\ \alpha_1 \ldots \alpha_k)}$ defined by $V(C, \eta)\ d_1 \ldots d_n = C\ d_1 \ldots d_n$, where $d_i \in \mathcal{D}_{\sigma_i\eta}$ $(i = 1, \ldots, n)$.
2. For $=$ with signature $a \rightarrow a \rightarrow \Omega$, $V(=, \{a/\alpha\})$ is the mapping from $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha \rightarrow \Omega}$ defined by

$$V(=, \{a/\alpha\})\ x\ y = \begin{cases} \top & \text{if } x = y \\ \bot & \text{otherwise.} \end{cases}$$

3. $V(\neg, \{\})$ is the mapping from $\mathcal{D}_\Omega$ into $\mathcal{D}_\Omega$ given by the following table.

x	$V(\neg, \{\})\ x$
$\top$	$\bot$
$\bot$	$\top$

4. $V(\wedge, \{\})$, $V(\vee, \{\})$, $V(\longrightarrow, \{\})$, and $V(\longleftrightarrow, \{\})$ are the mappings from $\mathcal{D}_\Omega$ into $\mathcal{D}_{\Omega\to\Omega}$ given by the following table.

x	y	$V(\wedge, \{\})\ x\ y$	$V(\vee, \{\})\ x\ y$	$V(\longrightarrow, \{\})\ x\ y$	$V(\longleftrightarrow, \{\})\ x\ y$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$

5. For Σ with signature $(a \to \Omega) \to \Omega$, $V(\Sigma, \{a/\alpha\})$ is the mapping from $\mathcal{D}_{\alpha\to\Omega}$ to $\mathcal{D}_\Omega$ which maps an element f of $\mathcal{D}_{\alpha\to\Omega}$ to $\top$ if f maps at least one element of $\mathcal{D}_\alpha$ to $\top$; otherwise, it maps f to $\bot$.
6. For Π with signature $(a \to \Omega) \to \Omega$, $V(\Pi, \{a/\alpha\})$ is the mapping from $\mathcal{D}_{\alpha\to\Omega}$ to $\mathcal{D}_\Omega$ which maps an element f of $\mathcal{D}_{\alpha\to\Omega}$ to $\top$ if f maps every element of $\mathcal{D}_\alpha$ to $\top$; otherwise, it maps f to $\bot$.

Definition 2.7.2 ensures that the connectives and quantifiers have their usual meaning and that data constructors have the free interpretation.

Definition 2.7.3. A *variable assignment* with respect to an interpretation $\langle\{\mathcal{D}_\alpha\}_{\alpha\in\mathfrak{S}^c},\ V\rangle$ is a mapping that maps each pair consisting of a variable and a closed type α to an element of $\mathcal{D}_\alpha$.

The meaning of a term t with respect to an interpretation, a variable assignment, and a closed type substitution can now be defined. This definition is rather standard except for the introduction of the closed type substitution. Because types can be polymorphic, it is necessary to specify the meaning of t with respect to each closed instantiation of the parameters in the relative types of the subterms of t (which includes the parameters in the type of t, of course). Normally it is only necessary to specify the instantiation of the parameters in the type of t to achieve this. However, as noted earlier, it is possible for a (proper) subterm of t to have a relative type containing a parameter that does not appear in the type of t. Hence the following definition.

Definition 2.7.4. Let t be a term. A *grounding* type substitution for t is closed type substitution whose domain includes the set of all parameters in the relative types of subterms of t.

Definition 2.7.5. Let t be a term of type α, η a grounding type substitution for t, $I \equiv \langle\{\mathcal{D}_\delta\}_{\delta\in\mathfrak{S}^c},\ V\rangle$ an interpretation, and φ a variable assignment with respect to I. Then the *denotation* $\mathcal{V}(t, \eta, I, \varphi)$ of t with respect to η, I, and φ is defined inductively as follows.

1. $\mathcal{V}(x, \eta, I, \varphi) = \varphi(x, a\eta)$, where x is a variable of type a.
2. $\mathcal{V}(C, \eta, I, \varphi) = V(C, \eta')$, where C is a constant and η' is η restricted to the parameters in the signature of C.
3. $\mathcal{V}(\lambda x.s, \eta, I, \varphi) =$ the function whose value for each $d \in \mathcal{D}_{\beta\eta}$ is $\mathcal{V}(s, \eta, I, \varphi')$, where $\lambda x.s$ has type $\beta \to \gamma$ and φ' is φ except $\varphi'(x, \beta\eta) = d$.

4. $\mathcal{V}((u\ v), \eta, I, \varphi) = \mathcal{V}(u, \xi\eta, I, \varphi)(\mathcal{V}(v, \xi\eta, I, \varphi))$, where ξ is the associated mgu for $(u\ v)$.
5. $\mathcal{V}((t_1, \ldots, t_n), \eta, I, \varphi) = (\mathcal{V}(t_1, \xi\eta, I, \varphi), \ldots, \mathcal{V}(t_n, \xi\eta, I, \varphi))$, where ξ is the associated mgu for $(t_1, \ldots, t_n)$.

One can show that $\mathcal{V}(t, \eta, I, \varphi) \in \mathcal{D}_{\alpha\eta}$, where t has type α. (See Exercise 2.16.) Furthermore, one can show that if η_1 and η_2 are grounding type substitutions for t that agree on the set of all parameters in the relative types of subterms of t, then $\mathcal{V}(t, \eta_1, I, \varphi) = \mathcal{V}(t, \eta_2, I, \varphi)$. (See Exercise 2.17.)

Example 2.7.7. Let the alphabet contain the nullary type constructor *Int*, the unary type constructor *List*, and the constants

$$concat : List\ a \rightarrow List\ a \rightarrow List\ a,$$

$$[] : List\ a,$$

$$\sharp : a \rightarrow List\ a \rightarrow List\ a,$$

as well as the integers. Let $\mathcal{D}_{List\ Int}$ be the set of finite lists of integers. Suppose the interpretation I includes assigning $V(concat, \{a/Int\})$ to be the function that concatenates lists of integers. Finally, let t be the term $concat\ []\ x$, η the grounding type substitution $\{a/Int\}$ for t, and φ a variable assignment with respect to I that includes $\varphi(x, List\ Int) = 1 \sharp 2 \sharp []$. Then $\mathcal{V}(t, \eta, I, \varphi) = 1 \sharp 2 \sharp []$.

If t is a closed term, then $\mathcal{V}(t, \eta, I, \varphi)$ is independent of φ, as the following proposition shows.

Proposition 2.7.1. *Let t be a term, η a grounding type substitution for t, I an interpretation, and φ_1 and φ_2 variable assignments with respect to I. Suppose that, for each free variable x of relative type β in t, $\varphi_1(x, \beta\eta) = \varphi_2(x, \beta\eta)$. Then $\mathcal{V}(t, \eta, I, \varphi_1) = \mathcal{V}(t, \eta, I, \varphi_2)$.*

Proof. The proof is by induction on the structure of t. If t is a variable or a constant, then the result is obvious.

Let t have the form $\lambda x.s$ and suppose that, for each free variable y of relative type β in t, $\varphi_1(x, \beta\eta) = \varphi_2(x, \beta\eta)$. Now $\mathcal{V}(t, \eta, I, \varphi_i)$ is the function from $\mathcal{D}_{\beta\eta}$ into $\mathcal{D}_{\gamma\eta}$ whose value for each $d \in \mathcal{D}_{\beta\eta}$ is $\mathcal{V}(s, \eta, I, \varphi'_i)$, where $\lambda x.s$ has type $\beta \rightarrow \gamma$ and φ'_i is φ_i except $\varphi'_i(x, \beta\eta) = d$, for $i = 1, 2$. By the induction hypothesis, $\mathcal{V}(s, \eta, I, \varphi'_1) = \mathcal{V}(s, \eta, I, \varphi'_2)$. Hence $\mathcal{V}(t, \eta, I, \varphi_1) = \mathcal{V}(t, \eta, I, \varphi_2)$.

Let t have the form $(u\ v)$ with associated mgu ξ and suppose that, for each free variable x of relative type β in t, $\varphi_1(x, \beta\eta) = \varphi_2(x, \beta\eta)$. Hence, for each free variable x of relative type δ in u, $\varphi_1(x, \delta\xi\eta) = \varphi_2(x, \delta\xi\eta)$, since $\delta\xi$ is the relative type of x in t. Similarly, for v. Now $\mathcal{V}((u\ v), \eta, I, \varphi_i) = \mathcal{V}(u, \xi\eta, I, \varphi_i)(\mathcal{V}(v, \xi\eta, I, \varphi_i))$, for $i = 1, 2$. By the induction hypothesis,

$\mathcal{V}(u, \xi\eta, I, \varphi_1) = \mathcal{V}(u, \xi\eta, I, \varphi_2)$ and $\mathcal{V}(v, \xi\eta, I, \varphi_1) = \mathcal{V}(v, \xi\eta, I, \varphi_2)$. Hence $\mathcal{V}((u\ v), \eta, I, \varphi_1) = \mathcal{V}((u\ v), \eta, I, \varphi_2)$.

Let t have the form $(t_1, \ldots, t_n)$ with associated mgu ξ and suppose that, for each free variable x of relative type β in t, $\varphi_1(x, \beta\eta) = \varphi_2(x, \beta\eta)$. Hence, for each free variable x of relative type δ in t_j, $\varphi_1(x, \delta\xi\eta) = \varphi_2(x, \delta\xi\eta)$, since $\delta\xi$ is the relative type of x in t, for $j = 1, \ldots, n$. Now $\mathcal{V}((t_1, \ldots, t_n), \eta, I, \varphi_i) = (\mathcal{V}(t_1, \xi\eta, I, \varphi_i), \ldots, \mathcal{V}(t_n, \xi\eta, I, \varphi_i))$, for $i = 1, 2$. By the induction hypothesis, $\mathcal{V}(t_j, \xi\eta, I, \varphi_1) = \mathcal{V}(t_j, \xi\eta, I, \varphi_2)$, for $j = 1, \ldots, n$. Hence $\mathcal{V}((t_1, \ldots, t_n), \eta, I, \varphi_1)$ $= \mathcal{V}(t_1, \ldots, t_n, \eta, I, \varphi_2)$. □

Definition 2.7.6. Let t be a formula, I an interpretation, and φ a variable assignment with respect to I.

1. φ *satisfies* t *in* I if $\mathcal{V}(t, \eta, I, \varphi) = \top$, for each grounding type substitution η for t.
2. t is *satisfiable in* I if there is a variable assignment which satisfies t in I.
3. t is *valid in* I if every variable assignment satisfies t in I.
4. t is *valid* if t is valid in every interpretation.
5. A *model* for a theory T is an interpretation in which each formula in T is valid.

Definition 2.7.7. A theory is *consistent* if it has a model.

Definition 2.7.8. Let T be a theory and t a formula. Then t is a *logical consequence* of T if t is valid in every model for T.

The last definition is a fundamental one. Typically for an application, there is a suitable theory describing the application and a distinguished model, called the *intended interpretation*, for the theory. Then, for a particular formula, there arises the question as to whether or not it is valid in the intended interpretation. To settle this, one establishes whether or not the formula is a logical consequence of the theory. If the formula is a logical consequence of the theory, then it is valid in all models of the theory and, therefore, in the intended interpretation. The usual method of establishing logical consequence is to find a proof of the formula, as discussed in the next section.

2.8 Proof Theory

The proof-theoretic aspects of the logic are now explored.

Definition 2.8.1. A *proof system* consists of a set of axioms and some rules of inference.

The axioms – which are formulas – are of two kinds: *logical* axioms that are valid in every interpretation and *proper* axioms that are valid in the intended interpretation for the application. The proper axioms together constitute a theory, in the sense of Definition 2.3.5. A rule of inference specifies whether a formula can be inferred from some set of formulas.

Definition 2.8.2. A *proof* of a formula t is a finite sequence of formulas the last of which is t such that each formula is either an axiom or can be inferred from preceding formulas in the sequence by an application of some rule of inference.

A *theorem* is a formula with a proof.

For the model theory and the proof theory to fit together in a harmonious way, there are several desirable properties. The first is soundness: each theorem should be valid in every model of the theory (consisting of the set of proper axioms). The second is completeness: each formula that is valid in every model of the theory should be a theorem.

There are various formulations of a suitable set of logical axioms for the non-polymorphic case; these could be adapted for the polymorphic case considered here. These axioms are not discussed further for the reasons given below.

A suitable inference rule is as follows:

From formulas t and $s = r$, where s is a subterm of t at occurrence o and $s \approx r$, infer $t[s/r]_o$.

More generally, one could merely require that s be α-equivalent to a subterm of t and everything would work well. By Proposition 2.4.7, it follows that $t[s/r]_o$ is a formula and $t[s/r]_o \approx t$.

Proposition 2.8.3 below shows that the inference rule is sound (under the assumption that, whatever are the logical axioms, they are valid in any interpretation). To prove this, two preliminary results are needed.

Proposition 2.8.1. *Let $s = t$ be a term, ξ the associated mgu for $((= s)\ t)$, I an interpretation, φ a variable assignment with respect to I, and η a grounding type substitution for $s = t$. Then $\xi\eta$ is a grounding type substitution for s and t, and $\mathcal{V}(s = t, \eta, I, \varphi) = \top$ iff $\mathcal{V}(s, \xi\eta, I, \varphi) = \mathcal{V}(t, \xi\eta, I, \varphi)$.*

Proof. It is clear that $\xi\eta$ is a grounding type substitution for s and t. Suppose that s is a term of type σ. Since $= : a \to a \to \Omega$, the associated mgu ζ for $(= s)$ is $\{a/\sigma\}$. Note that $\mathcal{V}(s, \zeta\xi\eta, I, \varphi) = \mathcal{V}(s, \xi\eta, I, \varphi)$, since a does not appear in the relative type of any subterm of s. Also $\mathcal{V}(=, \zeta\xi\eta, I, \varphi)$ is the identity relation on the domain $\mathcal{D}_{a\zeta\xi\eta}$. Thus

$$\begin{aligned}
&\mathcal{V}(s = t, \eta, I, \varphi) = \top \\
&\text{iff } \mathcal{V}((= s), \xi\eta, I, \varphi)\ (\mathcal{V}(t, \xi\eta, I, \varphi)) = \top \\
&\text{iff } (\mathcal{V}(=, \zeta\xi\eta, I, \varphi)\ (\mathcal{V}(s, \zeta\xi\eta, I, \varphi)))\ (\mathcal{V}(t, \xi\eta, I, \varphi)) = \top \\
&\text{iff } \mathcal{V}(s, \xi\eta, I, \varphi) = \mathcal{V}(t, \xi\eta, I, \varphi).
\end{aligned}$$

□

Proposition 2.8.2. *Let t be a term, s a subterm of t at occurrence o, r a term such that $r \precsim s$, I an interpretation, and φ a variable assignment with respect to I. Suppose that $\mathcal{V}(s = r, \zeta, I, \varphi) = \top$, for each grounding type substitution ζ for $s = r$. Let ξ be the associated mgu for $((= \ t[s/r]_o)\ t)$. Then, for each grounding type substitution η for t, there exists a type substitution ω such that $\xi\eta\omega$ is a grounding type substitution for $t[s/r]_o$ and $\mathcal{V}(t, \eta, I, \varphi) = \mathcal{V}(t[s/r]_o, \xi\eta\omega, I, \varphi)$.*

Proof. Note that, by Proposition 2.4.6, $((= \ t[s/r]_o)\ t)$ is a term and $t[s/r]_o \precsim t$. The proof is by induction on the length n of o.

Suppose first that $n = 0$. Thus t is s and $t[s/r]_o$ is r. Since $r \precsim s$, it can be assumed without loss of generality that ξ does not act on the parameters in the relative types of subterms of s. Let η be a grounding type substitution for s and ω any grounding type substitution for $s = r$. Then $\eta\omega$ is a grounding type substitution for $s = r$. (The type substitution ω is used just to pick up any parameters in relative types of subterms of r that are not instantiated by η.) By Proposition 2.8.1, $\mathcal{V}(s, \xi\eta\omega, I, \varphi) = \mathcal{V}(r, \xi\eta\omega, I, \varphi)$. However, $\mathcal{V}(s, \xi\eta\omega, I, \varphi) = \mathcal{V}(s, \eta\omega, I, \varphi)$, since ξ does not act on the parameters in the relative types of subterms of s, and $\mathcal{V}(s, \eta\omega, I, \varphi) = \mathcal{V}(s, \eta, I, \varphi)$, since $\eta\omega$ and η agree on the set of all parameters in the relative types of subterms of s. Thus $\mathcal{V}(s, \eta, I, \varphi) = \mathcal{V}(r, \xi\eta\omega, I, \varphi)$, as required.

Suppose next that the result holds for occurrences of length n and o has length $n + 1$. Then t has the form $\lambda x.s$, $(u\ v)$, or $(t_1, \ldots, t_n)$. Consider the case when t has the form $(u\ v)$ and $o = 1o'$, for some o'. Let α be the associated mgu for $(u\ v)$, β the associated mgu for $(u[s/r]_{o'}\ v)$, and π the associated mgu for $((= \ u[s/r]_{o'})\ u)$. It can be assumed without loss of generality that π does not act on any parameters in the relative types of subterms of v. One can show that $\pi\alpha$ and $\beta\xi$ agree on the set of all parameters that appear in relative types of subterms of $u[r/s]_{o'}$; similarly, for v. Let η be a grounding type substitution for $(u\ v)$. Then $\alpha\eta$ is a grounding type substitution for u and, by the induction hypothesis, there exists a type substitution ω such that $\mathcal{V}(u, \alpha\eta, I, \varphi) = \mathcal{V}(u[s/r]_{o'}, \pi\alpha\eta\omega, I, \varphi)$. Note that $\mathcal{V}(v, \alpha\eta, I, \varphi) = \mathcal{V}(v, \alpha\eta\omega, I, \varphi)$, since $\alpha\eta$ and $\alpha\eta\omega$ agree on the set of all parameters in the relative types of subterms of v. Thus

$$
\begin{aligned}
\mathcal{V}((u\ v), \eta, I, \varphi) &= \mathcal{V}(u, \alpha\eta, I, \varphi)(\mathcal{V}(v, \alpha\eta, I, \varphi)) \\
&= \mathcal{V}(u[s/r]_{o'}, \pi\alpha\eta\omega, I, \varphi)(\mathcal{V}(v, \alpha\eta, I, \varphi)) \\
&= \mathcal{V}(u[s/r]_{o'}, \pi\alpha\eta\omega, I, \varphi)(\mathcal{V}(v, \alpha\eta\omega, I, \varphi)) \\
&= \mathcal{V}(u[s/r]_{o'}, \pi\alpha\eta\omega, I, \varphi)(\mathcal{V}(v, \pi\alpha\eta\omega, I, \varphi)) \\
&= \mathcal{V}(u[s/r]_{o'}, \beta\xi\eta\omega, I, \varphi)(\mathcal{V}(v, \beta\xi\eta\omega, I, \varphi)) \\
&= \mathcal{V}((u[s/r]_{o'}\ v), \xi\eta\omega, I, \varphi) \\
&= \mathcal{V}((u\ v)[s/r]_o, \xi\eta\omega, I, \varphi).
\end{aligned}
$$

The argument is similar when $o = 2o'$.

The other two cases when t has the form $\lambda x.s$ or $(t_1, \ldots, t_n)$ are similar. □

Proposition 2.8.3. *Every theorem is a logical consequence of the theory consisting of the set of proper axioms.*

Proof. Let t be a formula, s a subterm of t at occurrence o, r a term such that $s \approx r$, I an interpretation, and φ a variable assignment with respect to I. Suppose that, for each grounding type substitution ζ for $s = r$, $\mathcal{V}(s = r, \zeta, I, \varphi) = \top$ and, for each grounding type substitution υ for t, $\mathcal{V}(t, \upsilon, I, \varphi) = \top$. It suffices to prove that, for each grounding type substitution η for $t[s/r]_o$, $\mathcal{V}(t[s/r]_o, \eta, I, \varphi) = \top$.

Note first that $(t[s/r]_o)[r/s]_o$ is t. Let ξ be the associated mgu for $((= \ (t[s/r]_o)[r/s]_o) \ t[s/r]_o)$. According to Proposition 2.8.2, there exists a type substitution ω such that $\mathcal{V}(t[s/r]_o, \eta, I, \varphi) = \mathcal{V}((t[s/r]_o)[r/s]_o, \xi\eta\omega, I, \varphi)$. But $\mathcal{V}((t[s/r]_o)[r/s]_o, \xi\eta\omega, I, \varphi) = \mathcal{V}(t, \xi\eta\omega, I, \varphi)$ and $\mathcal{V}(t, \xi\eta\omega, I, \varphi) = \top$, by the assumptions. Hence the result. □

One could now proceed to give a complete account of a conventional proof theory for the logic. This essentially requires extending the non-polymorphic development to the polymorphic case. (See Exercise 2.21.) However, this proof theory has little importance for the applications to machine learning, so instead I concentrate in Chap. 5 on the problem of defining a suitable operational behaviour for programs in declarative programming languages whose programs are certain kinds of equational theories.

Bibliographical Notes

The logic of this chapter can be traced back to Church's simple theory of types [11]. The appropriate model theory for type theory and the completeness result was given subsequently by Henkin in [34]. A detailed account of the proof theory and model theory for type theory is given in [1]. See also [90]. Categorical accounts of the logic are in [2] and [46].

The form of polymorphism employed in this chapter has a long history. In 1958, the concept of a polymorphic type appeared in combinatory logic [16], being called there the 'functional character' of an expression. In 1969, Hindley [36] introduced the idea of a principal type schema, which is the most general polymorphic type of an expression. Nearly ten years later, in a highly influential paper [60], Milner independently rediscovered these ideas and also introduced the first practical polymorphic type checker. The Hindley–Milner type system, as it has become known, has been widely used in functional and logic programming languages.

A highly recommended reference on (first-order) unification is [48], which provides an introduction to the basic ideas on equation solving and unification, and presents a large number of useful results with their proofs. A brief account of unification is given in [52]. There are various algorithms for unifying a set of equations about first-order terms and, therefore, about types. See [48] and [52] for two of these. Higher-order unification is discussed in [90].

Exercises

2.1 Let $\alpha \in \mathfrak{S}$. Prove that exactly one of the following holds.

1. α is a parameter.
2. α has the form $T\ \alpha_1 \ldots \alpha_k$, where $T \in \mathfrak{T}$ and $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}$.
3. α has the form $\beta \to \gamma$, where $\beta, \gamma \in \mathfrak{S}$.
4. α has the form $\alpha_1 \times \cdots \times \alpha_n$, where $\alpha_1, \ldots, \alpha_n \in \mathfrak{S}$.

2.2 Let $\alpha \in \mathfrak{S}^c$. Prove that exactly one of the following holds.

1. α has the form $T\ \alpha_1 \ldots \alpha_k$, where $T \in \mathfrak{T}$ and $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c$.
2. α has the form $\beta \to \gamma$, where $\beta, \gamma \in \mathfrak{S}^c$.
3. α has the form $\alpha_1 \times \cdots \times \alpha_n$, where $\alpha_1, \ldots, \alpha_n \in \mathfrak{S}^c$.

2.3 Prove that $\alpha(\mu\nu) = (\alpha\mu)\nu$, for any type α and type substitutions μ and ν.

2.4 Prove that if σ is an mgu for a set E of equations and α is invertible, then $\sigma\alpha$ is also an mgu for E.

2.5 Give an inductive definition of the instance of a type by a type substitution. Hence prove that if α is a type and μ is a type substitution, then $\alpha\mu$ is a type.

2.6 Prove that a type substitution is invertible iff it is a permutation of parameters.

2.7 Let μ_1 and μ_2 be mgus. Prove that μ_1 and μ_2 are mgus of the same set of equations about types iff $\mu_1 = \mu_2\alpha$, for some invertible type substitution α.

2.8 Prove that a type σ is a variant of a type τ iff there exist type substitutions θ and φ such that $\sigma\theta = \tau$ and $\tau\varphi = \sigma$.

2.9 Give an example to show what undesirable consequences can follow if parameters are not standardised apart in Parts 4 and 5 of Definition 2.3.2.

2.10 This problem provides a 'bottom-up' definition of $\mathfrak{L}$.

(i) Complete the following inductive definition of $\{\mathfrak{L}_m\}_{m\in\mathbb{N}}$ so that Parts (ii) and (iii) below make sense by filling in the missing conditions and making any other necessary modifications.
[Hint: annotated terms will be needed.]

$$
\begin{aligned}
\mathfrak{L}_0 &= \{t \mid t \text{ is a variable or a constant}\} \\
\mathfrak{L}_{m+1} &= \mathfrak{L}_m \\
&\quad \cup \{\lambda x.t \mid t \in \mathfrak{L}_m\} \\
&\quad \cup \{(s\ t) \mid s, t \in \mathfrak{L}_m \text{ and some condition}\} \\
&\quad \cup \{(t_1, \dots, t_n) \mid t_1, \dots, t_n \in \mathfrak{L}_m \text{ and some condition}\}.
\end{aligned}
$$

(ii) Prove that $\mathfrak{L}_m \subseteq \mathfrak{L}_{m+1}$, for $m \in \mathbb{N}$.
(iii) Prove that $\mathfrak{L} = \bigcup_{m\in\mathbb{N}} \mathfrak{L}_m$.

2.11 Prove that if r is the subterm of term s at occurrence o' and s is the subterm of term t at occurrence o, then r is the subterm of t at occurrence oo'.

2.12 Prove that if s and t are terms such that $s \precsim t$ and $t \precsim s$, then $s \approx t$.

2.13 Prove Proposition 2.5.3.

2.14 Let t be a term having type σ, $x_1, \dots, x_n$ be free variables in t, and $\theta \equiv \{x_1/t_1, \dots, x_n/t_n\}$ be an idempotent term substitution. Suppose that $U_{t,\theta} \cup V_{t,\theta}$ has mgu φ. Prove that $(\lambda x_n.(\dots(\lambda x_1.t\ t_1)\dots)\ t_n)$ is a term of type $\sigma\varphi$ and, if y is a free variable in $(\lambda x_n.(\dots(\lambda x_1.t\ t_1)\dots)\ t_n)$ and y has relative type δ in t or some t_i, then y has relative type $\delta\varphi$ in $(\lambda x_n.(\dots(\lambda x_1.t\ t_1)\dots)\ t_n)$.

2.15 Let t be a term having type σ, $x_1, \dots, x_n$ be free variables in t, and $\theta \equiv \{x_1/t_1, \dots, x_n/t_n\}$ be an idempotent term substitution. Suppose that $(\lambda x_n.(\dots(\lambda x_1.t\ t_1)\dots)\ t_n)$ is a term. Prove that $U_{t,\theta} \cup V_{t,\theta}$ is unifiable.

2.16 Let t be a term of type α, η a grounding type substitution for t, $I \equiv \langle\{\mathcal{D}_\delta\}_{\delta\in\mathfrak{S}^c}, V\rangle$ an interpretation, and φ a variable assignment with respect to I. Prove that $\mathcal{V}(t, \eta, I, \varphi) \in \mathcal{D}_{\alpha\eta}$.

2.17 Let t be a term of type α, I an interpretation, and φ a variable assignment with respect to I. Suppose that η_1 and η_2 are grounding type substitutions for t that agree on the set of all parameters in the relative types of subterms of t. Prove that $\mathcal{V}(t, \eta_1, I, \varphi) = \mathcal{V}(t, \eta_2, I, \varphi)$.

2.18 Complete the details of the proof of Proposition 2.8.2.

2.19 (Open problem) Prove or disprove the following conjecture.
Let t be a term, $x_1, \dots, x_n$ variables that occur freely exactly once in t, where x_i has relative type ϱ_i in t, for $i = 1, \dots, n$, and $\theta \equiv \{x_1/t_1, \dots, x_n/t_n\}$ an idempotent term substitution such that $U_{t,\theta} \cup V_{t,\theta}$ has mgu α, say. Then, for every interpretation I, grounding type substitution η for $t\theta$, and variable assignment φ with respect to I, it follows that $\alpha\eta$ is a grounding type substitution for t and $\mathcal{V}(t\theta, \eta, I, \varphi) = \mathcal{V}(t, \alpha\eta, I, \varphi')$, where φ' is φ except $\varphi'(x_i, \varrho_i\alpha\eta) = \mathcal{V}(t_i, \alpha\eta, I, \varphi)$, for $i = 1, \dots, n$.

2.20 (Open problem) Prove or disprove the following conjecture.
Let $s = t$ be a term, $x_1, \ldots, x_n$ variables that occur freely exactly once in s, and $\theta \equiv \{x_1/t_1, \ldots, x_n/t_n\}$ an idempotent term substitution such that $t \precsim s$ and $s\theta = t\theta$ is a term. Let I be an interpretation and φ a variable assignment with respect to I such that $\mathcal{V}(s = t, \zeta, I, \varphi) = \top$, for each grounding type substitution ζ for $s = t$. Then, for each grounding type substitution η for $s\theta$, there exists a grounding type substitution ν for $t\theta$ such that $\mathcal{V}(s\theta, \eta, I, \varphi) = \mathcal{V}(t\theta, \nu, I, \varphi)$.

2.21 (Open problem) Extend the presentation of the proof system in [1, Chap. 5] to the polymorphic case. The extended presentation should include a suitably generalised form of Henkin's completeness theorem.

3. Individuals

In this chapter, the application of the logic to knowledge representation is studied. The main idea is the identification of a class of terms, called *basic terms*, suitable for representing individuals in diverse applications. For example, this class is suitable for machine-learning applications. From a (higher-order) programming language perspective, basic terms are data values. The most interesting aspect of the class of basic terms is that it includes certain abstractions and therefore is wider than is normally considered for knowledge representation. These abstractions allow one to model sets, multisets, and data of similar types, in a direct way. Of course, there are other ways of introducing (extensional) sets, multisets, and so on, without using abstractions. For example, one can define abstract data types or one can introduce data constructors with special equality theories. The primary advantage of the approach adopted here is that one can define these abstractions *intensionally*, as shown in Chap. 5. Techniques for defining metrics and kernels on basic terms are also investigated in this chapter.

The definition of basic terms is given in several stages: first normal terms are defined, then an equivalence relation on normal terms is defined, and, finally, basic terms as distinguished representatives of equivalence classes are defined. To define normal terms, the concept of a default term is needed, so the development starts there.

3.1 Default Terms

Before getting down to the first step of giving the definition of normal terms, some motivation will be helpful. How should a (finite) set or multiset be represented? First, advantage is taken of the higher-order nature of the logic to identify sets and their characteristic functions; that is, sets are viewed as predicates. With this approach, an obvious representation of sets uses the connectives, so that $\lambda x.(x = 1) \vee (x = 2)$ is the representation of the set $\{1, 2\}$. This kind of representation works well for sets. But the connectives are, of course, not available for multisets, so something more general is needed. An alternative representation for the set $\{1, 2\}$ is the term

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

and this idea generalises to multisets and similar abstractions. For example,

$$\lambda x.\mathit{if}\ x = A\ \mathit{then}\ 42\ \mathit{else}\ \mathit{if}\ x = B\ \mathit{then}\ 21\ \mathit{else}\ 0$$

is the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus abstractions of the form

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

are adopted to represent (extensional) sets, multisets, and so on.

However, before giving the definition of a normal term, some attention has to be paid to the term s_0 in the previous expression. The reason is that s_0 in this abstraction is usually a very specific term. For example, for finite sets, s_0 is $\bot$ and for finite multisets, s_0 is 0. For this reason, the concept of a default term is now introduced. The intuitive idea is that, for each closed type, there is a (unique) default term such that each abstraction having that type as codomain takes the default term as its value for all but a finite number of points in the domain, that is, s_0 is the default value. The choice of default term depends on the particular application but, since sets and multisets are so useful, one would expect the set of default terms to include $\bot$ and 0. However, there could also be other types for which a default term is needed.

For each type constructor T, I assume there is chosen a unique *default data constructor* C such that C has signature of the form $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. For example, for Ω, the default data constructor could be $\bot$, for *Int*, the default data constructor could be 0, and for *List*, the default data constructor could be [].

Definition 3.1.1. The set of *default terms*, $\mathfrak{D}$, is defined inductively as follows.

1. If C is a default data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{D}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{D}$.
2. If $t \in \mathfrak{D}$ and $x \in \mathfrak{V}$, then $\lambda x.t \in \mathfrak{D}$.
3. If $t_1, \ldots, t_n \in \mathfrak{D}$ $(n \geq 0)$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{D}$.

Note 3.1.1. To be precise, the meaning of the previous inductive definition is that $\mathfrak{D}$ is the intersection of all sets of terms each satisfying (appropriately reworded versions of) Conditions 1 to 3. A suitable universe for the construction is the set $\mathfrak{L}$ of all terms. In particular, $\mathfrak{D} \subseteq \mathfrak{L}$.

To prove properties of default terms, one can employ the following *principle of induction on the structure of default terms.*

Proposition 3.1.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{D}$ satisfying the following conditions.*

1. *If C is a default data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{X}$.*

2. *If* $t \in \mathfrak{X}$ *and* $x \in \mathfrak{V}$, *then* $\lambda x.t \in \mathfrak{X}$.
3. *If* $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \geq 0)$ *and* $(t_1, \ldots, t_n) \in \mathfrak{L}$, *then* $(t_1, \ldots, t_n) \in \mathfrak{X}$.

Then $\mathfrak{X} = \mathfrak{D}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a default term, it follows that $\mathfrak{D} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{D}$. □

Proposition 3.1.2. *Each default term is closed.*

Proof. The proof is by induction on the structure of default terms. Let $\mathfrak{X}$ be the set of all default terms that are closed.

Suppose that C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{X}$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$. Then $C\ t_1 \ldots t_n \in \mathfrak{D}$ and $C\ t_1 \ldots t_n$ is closed. Thus $C\ t_1 \ldots t_n \in \mathfrak{X}$.

Suppose next that $t \in \mathfrak{X}$ and $x \in \mathfrak{V}$. Then $\lambda x.t \in \mathfrak{D}$ and $\lambda x.t$ is closed. Thus $\lambda x.t \in \mathfrak{X}$.

Finally, suppose that $t_1, \ldots, t_n \in \mathfrak{X}$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$. Then $(t_1, \ldots, t_n) \in \mathfrak{D}$ and $(t_1, \ldots, t_n)$ is closed. Thus $(t_1, \ldots, t_n) \in \mathfrak{X}$.

Hence $\mathfrak{X} = \mathfrak{D}$, by Proposition 3.1.1. □

Since each default term is closed, it follows that a default term $\lambda x.t$ has a type of the form $a \to \beta$, for some parameter a, since t is closed and so x is not free in t.

It will be convenient to gather together all default terms that have a type more general than some specific closed type.

Definition 3.1.2. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{D}_\alpha = \{t \in \mathfrak{D} \mid t$ has type more general than $\alpha\}$.

Note that $\mathfrak{D} = \bigcup_{\alpha \in \mathfrak{S}^c} \mathfrak{D}_\alpha$. However, the $\mathfrak{D}_\alpha$ may not be disjoint. For example, if the alphabet includes the type constructor *List* and [] is the default data constructor for *List*, then $[] \in \mathfrak{D}_{List\ \alpha}$, for each closed type α. Furthermore, $\mathfrak{D}_\alpha$ may be empty, for some α.

Example 3.1.1. Assume the alphabet contains just the nullary type constructors M and N (in addition to *1* and Ω) and the data constructors $F : M \to N$ and $G : N \to M$. (Recall that each type constructor must have an associated data constructor.) Let G be the default data constructor for M. Then there are no *closed* terms of type M and hence $\mathfrak{D}_M$ is empty.

Note 3.1.2. It is tempting to try to prove results about default terms in the various $\mathfrak{D}_\alpha$ by using induction on the structure of *closed types*. This will work for closed types of the form $\alpha \to \beta$ or $\alpha_1 \times \cdots \times \alpha_n$ because the types of the (top-level) subterms of the corresponding term are subtypes of $\alpha \to \beta$ or $\alpha_1 \times \cdots \times \alpha_n$. But it does not generally work for types of the form $T\ \alpha_1 \ldots \alpha_k$ because the types of the (top-level) subterms of the corresponding term are not likely to be related to $\alpha_1, \ldots, \alpha_k$, and thus the

obvious induction hypothesis is not useful. (One exception is the proof of Proposition 3.1.4 for which the assumption of the proposition means that the induction hypothesis is not needed for types of the form $T\ a_1 \ldots a_k$.) This is because, if a term has a (top-level) data constructor with signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$, the types of the (top-level) subterms are related to $\sigma_1, \ldots, \sigma_n$. (See Proposition 3.1.3.) This explains why some of the proofs may appear, at first sight, to be more complicated than necessary. The same comment applies to normal and basic terms introduced below. (See, for example, Proposition 3.2.5.)

The next result gives some detail about the structure of default terms.

Proposition 3.1.3.

1. *Let T be a type constructor and C the default data constructor for T, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. Let $t \in \mathfrak{D}_{T\ \alpha_1 \ldots \alpha_n}$ and $\xi = \{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$. Then $t = C\ t_1 \ldots t_n$, where $t_i \in \mathfrak{D}_{\sigma_i \xi}$, for $i = 1, \ldots, n$.*
2. *Let $t \in \mathfrak{D}_{\beta \to \gamma}$. Then $t = \lambda x.u$, where $u \in \mathfrak{D}_\gamma$.*
3. *Let $t \in \mathfrak{D}_{\alpha_1 \times \cdots \times \alpha_n}$. Then $t = (t_1, \ldots, t_n)$, where $t_i \in \mathfrak{D}_{\alpha_i}$, for $i = 1, \ldots, n$.*

Proof. 1. By the uniqueness of C, t has the form $C\ t_1 \ldots t_n$, where $t_1, \ldots, t_n \in \mathfrak{D}$. Suppose that t_i has type τ_i, for $i = 1, \ldots, n$. Let θ be an mgu of $\{\sigma_1 = \tau_1, \ldots, \sigma_n = \tau_n\}$. Thus $C\ t_1 \ldots t_n$ has type $(T\ a_1 \ldots a_n)\theta$. Now $C\ t_1 \ldots t_n \in \mathfrak{D}_{T\ \alpha_1 \ldots \alpha_n}$. Hence there exists γ such that $(T\ a_1 \ldots a_n)\theta\gamma = (T\ \alpha_1 \ldots \alpha_n)$. Then $\tau_i\theta\gamma = \sigma_i\theta\gamma = \sigma_i\xi$, for $i = 1, \ldots, n$. That is, $t_i \in \mathfrak{D}_{\sigma_i\xi}$, for $i = 1, \ldots, n$.

2. By the definition of default terms, $t = \lambda x.u$, for some $x \in \mathfrak{V}$ and $u \in \mathfrak{D}$. Since $\lambda x.u \in \mathfrak{D}_{\beta \to \gamma}$, $\lambda x.u$ has type of the form $a \to \sigma$, where σ is the type of u and σ is more general than γ. Hence $u \in \mathfrak{D}_\gamma$.

3. By the definition of default terms, $t = (t_1, \ldots, t_n)$, where $t_i \in \mathfrak{D}$, for $i = 1, \ldots, n$. Since $(t_1, \ldots, t_n) \in \mathfrak{D}_{\alpha_1 \times \cdots \times \alpha_n}$, each t_i has a type more general than α_i and hence $t_i \in \mathfrak{D}_{\alpha_i}$, for $i = 1, \ldots, n$. □

The next definition will provide a necessary and sufficient condition for each $\mathfrak{D}_\alpha$ to be non-empty.

Definition 3.1.3. A data constructor C having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ is *full* if, for all $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c$, it is true that $\mathfrak{D}_{\sigma_i\xi} \neq \emptyset$, for $i = 1, \ldots, n$, where $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$.

In particular, if $n = 0$, then C is full.

Proposition 3.1.4. *$\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$, iff each default data constructor is full.*

Proof. Suppose first that each default data constructor is full. The proof that $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$, proceeds by induction on the structure of closed types.

Let $\alpha \in \mathfrak{S}^c$ have the form $T\ \alpha_1 \ldots \alpha_k$ and suppose C is the default data constructor associated with T having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. Let $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Since C is full, there exist $t_i \in \mathfrak{D}_{\sigma_i \xi}$, for $i = 1, \ldots, n$. Then $C\ t_1 \ldots t_n \in \mathfrak{D}_{T\ \alpha_1 \ldots \alpha_k}$. (Note that this part does not need the induction hypothesis.)

Let $\alpha = \beta \to \gamma$. By the induction hypothesis, there exists $t \in \mathfrak{D}_\gamma$. Thus $\lambda x.t \in \mathfrak{D}_\alpha$.

Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. By the induction hypothesis, there exists $t_i \in \mathfrak{D}_{\alpha_i}$, for $i = 1, \ldots, n$. Thus $(t_1, \ldots, t_n) \in \mathfrak{D}_\alpha$.

Conversely, suppose that there exists a default data constructor C having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ that is not full. Hence there exist $\alpha_1, \ldots, \alpha_k \in \mathfrak{S}^c$ such that $\mathfrak{D}_{\sigma_{i_0} \xi} = \emptyset$, for some $i_0 \in \{1, \ldots, n\}$, where $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. Consequently, $\mathfrak{D}_{T\ \alpha_1 \ldots \alpha_k} = \emptyset$. □

However, if it exists, one can show that the default term for each closed type is unique.

Proposition 3.1.5. *For each $\alpha \in \mathfrak{S}^c$, there exists at most one default term in $\mathfrak{D}_\alpha$.*

Proof. Put $\mathfrak{X} = \{t \in \mathfrak{D} \mid$ if $s \in \mathfrak{D}$ and $s, t \in \mathfrak{D}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, then $s = t\}$. It suffices to show that $\mathfrak{X} = \mathfrak{D}$. For this, the three conditions of Proposition 3.1.1 are established.

Let C be a default data constructor having signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \geq 0)$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$. It must be shown that $C\ t_1 \ldots t_n \in \mathfrak{X}$. For this, suppose that $s \in \mathfrak{D}$ and $s, C\ t_1 \ldots t_n \in \mathfrak{D}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. Suppose that $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$. Put $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. To show that $C\ t_1 \ldots t_n \in \mathfrak{X}$, it suffices to show that $s = C\ t_1 \ldots t_n$. Now, by Proposition 3.1.3, $s = C\ s_1 \ldots s_n$, where $s_i \in \mathfrak{D}_{\sigma_i \xi}$, for $i = 1, \ldots, n$. Also, by Proposition 3.1.3, $t_i \in \mathfrak{D}_{\sigma_i \xi}$, for $i = 1, \ldots, n$. Since $t_i \in \mathfrak{X}$, it follows that $s_i = t_i$, for $i = 1, \ldots, n$, and so $s = C\ t_1 \ldots t_n$. Thus $C\ t_1 \ldots t_n \in \mathfrak{X}$, as required.

Suppose that $t \in \mathfrak{X}$ and $x \in \mathfrak{V}$. It must be shown that $\lambda x.t \in \mathfrak{X}$. Thus suppose that $s \in \mathfrak{D}$ and $s, \lambda x.t \in \mathfrak{D}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. Suppose that $\alpha = \beta \to \gamma$, for some β and γ. It suffices to show that $s = \lambda x.t$. Now, by Proposition 3.1.3, $s = \lambda x.u$, for some $u \in \mathfrak{D}_\gamma$. Also, by Proposition 3.1.3, $t \in \mathfrak{D}_\gamma$. Since $t \in \mathfrak{X}$, it follows that $u = t$ and so $s = \lambda x.t$. Thus $\lambda x.t \in \mathfrak{X}$, as required.

Suppose that $t_1, \ldots, t_n \in \mathfrak{X}$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$. It must be shown that $(t_1, \ldots, t_n) \in \mathfrak{X}$. Thus, suppose that $s \in \mathfrak{D}$ and $s, (t_1, \ldots, t_n) \in \mathfrak{D}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. Suppose that $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$. It suffices to show that $s = (t_1, \ldots, t_n)$. Now, by Proposition 3.1.3, $s = (s_1, \ldots, s_n)$, where $s_i \in \mathfrak{D}_{\alpha_i}$, for $i = 1, \ldots, n$. Also, by Proposition 3.1.3, $t_i \in \mathfrak{D}_{\alpha_i}$, for

$i = 1, \ldots, n$. Since $t_i \in \mathfrak{X}$, it follows that $s_i = t_i$, for $i = 1, \ldots, n$, and so $s = (t_1, \ldots, t_n)$. Thus $(t_1, \ldots, t_n) \in \mathfrak{X}$, as required.

Since all three conditions of Proposition 3.1.1 have now been established, it follows that $\mathfrak{X} = \mathfrak{D}$ and the result is proved. □

In the second part of the previous proof it may happen that s and t have different bound variables. This is an unimportant difference, that is, I regard identity of terms as being 'identity up to α-conversion'. See Note 3.4.1 below.

It follows from Propositions 3.1.4 and 3.1.5 that, if each default data constructor is full, then $\mathfrak{D}_\alpha$ is a singleton set, for each $\alpha \in \mathfrak{S}^c$.

Proposition 3.1.5 shows that choosing $\sharp$ as the default constructor for *List* is a bad idea. The reason is that, with this choice, $\mathfrak{D}_{List\ \alpha} = \emptyset$, for each $\alpha \in \mathfrak{S}^c$. For suppose $t \in \mathfrak{D}_{List\ \alpha}$, for some fixed α. Then t has the form $\sharp\ h\ b$, for some $h \in \mathfrak{D}_\alpha$ and $b \in \mathfrak{D}_{List\ \alpha}$. The latter fact contradicts the uniqueness of t.

A bottom-up characterisation of default terms will be needed.

Definition 3.1.4. Define $\{\mathfrak{D}_m\}_{m \in \mathbb{N}}$ inductively as follows.

$$
\begin{aligned}
\mathfrak{D}_0 &= \{C \mid C \text{ is a nullary default data constructor}\} \\
\mathfrak{D}_{m+1} &= \{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \text{ is a default data constructor of arity } n, \\
&\qquad\qquad\qquad\qquad\qquad\qquad t_i \in \mathfrak{D}_m\ (1 \leq i \leq n),\ n \in \mathbb{N}\} \\
&\quad \cup \{\lambda x.t \in \mathfrak{L} \mid t \in \mathfrak{D}_m\} \\
&\quad \cup \{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_i \in \mathfrak{D}_m\ (1 \leq i \leq n), n \in \mathbb{N}\}.
\end{aligned}
$$

Proposition 3.1.6.

1. $\mathfrak{D}_m \subseteq \mathfrak{D}_{m+1}$, *for* $m \in \mathbb{N}$.
2. $\mathfrak{D} = \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$.

Proof. 1. This part is an easy induction argument.

2. First, I show that $\bigcup_{m \in \mathbb{N}} \mathfrak{D}_m \subseteq \mathfrak{D}$. To prove this, it suffices to show by induction that $\mathfrak{D}_m \subseteq \mathfrak{D}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{D}_0 \subseteq \mathfrak{D}$. Suppose next that $\mathfrak{D}_m \subseteq \mathfrak{D}$. It then follows from the definitions of $\mathfrak{D}_{m+1}$ and $\mathfrak{D}$ that $\mathfrak{D}_{m+1} \subseteq \mathfrak{D}$.

Now I show that $\mathfrak{D} \subseteq \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$. For this, it suffices to show that $\bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{D}$ (since $\mathfrak{D}$ is the smallest such set). Suppose that C is a default data constructor, $t_1, \ldots, t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$, and $C\ t_1 \ldots t_n \in \mathfrak{L}$. Since the $\mathfrak{D}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \ldots, t_n \in \mathfrak{D}_p$. Hence $C\ t_1 \ldots t_n \in \mathfrak{D}_{p+1}$ and so $C\ t_1 \ldots t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$. Similar arguments show that $\bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$ satisfies Conditions 2 and 3. □

Proposition 3.1.7. *If the set* $\mathfrak{T}$ *of type constructors is countable, then* $\mathfrak{D}$ *is countable.*

Proof. Since $\mathfrak{T}$ is countable and, for every type constructor, there is associated a unique default data constructor, the set of default data constructors is also countable. Since $\mathfrak{D} = \bigcup_{m \in \mathbb{N}} \mathfrak{D}_m$, it suffices to show by induction that $\mathfrak{D}_m$ is countable, for $m \in \mathbb{N}$. However, this follows easily from the definition of $\{\mathfrak{D}_m\}_{m \in \mathbb{N}}$. □

3.2 Normal Terms

Now normal terms can be defined. In the following, $\lambda x.s_0$ is regarded as the special case of

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

when $n = 0$.

Definition 3.2.1. The set of *normal terms*, $\mathfrak{N}$, is defined inductively as follows.

1. If C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{N}$ $(n \in \mathbb{N})$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{N}$.
2. If $t_1, \ldots, t_n \in \mathfrak{N}$, $s_1, \ldots, s_n \in \mathfrak{N}$ $(n \in \mathbb{N})$, $s_0 \in \mathfrak{D}$ and

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

   then

   $$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}.$$

3. If $t_1, \ldots, t_n \in \mathfrak{N}$ $(n \in \mathbb{N})$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{N}$.

Note 3.2.1. As usual, the meaning of the previous inductive definition is that $\mathfrak{N}$ is the intersection of all sets of terms each satisfying (appropriately reworded versions of) Conditions 1 to 3. A suitable universe for the construction is the set $\mathfrak{L}$ of all terms. In particular, $\mathfrak{N} \subseteq \mathfrak{L}$.

Part 1 of the definition of the set of normal terms states, in particular, that individual natural numbers, integers, and so on, are normal terms. Also a term formed by applying a data constructor to (all of) its arguments, each of which is a normal term, is a normal term. As an example of this, consider the following declarations of the data constructors *Circle* and *Rectangle*.

$$\mathit{Circle} : \mathit{Float} \to \mathit{Shape}$$
$$\mathit{Rectangle} : \mathit{Float} \to \mathit{Float} \to \mathit{Shape}.$$

Then (*Circle* 7.5) and (*Rectangle* 42.0 21.3) are normal terms of type *Shape*. However, (*Rectangle* 42.0) is not a normal term as not all arguments to

Rectangle are given. Normal terms coming from Part 1 of the definition are called *normal structures* and always have a type of the form $T\ \alpha_1 \ldots \alpha_k$.

The abstractions formed in Part 2 of the definition are 'almost constant' abstractions since they take the default term s_0 as value for all except a finite number of points in the domain. (The term s_0 is called the *default value* for the abstraction.) They are called *normal abstractions* and always have a type of the form $\beta \to \gamma$. This class of abstractions includes useful data types such as (finite) sets and multisets. More generally, normal abstractions can be regarded as lookup tables, with s_0 as the value for items not in the table.

Part 3 of the definition of normal terms just states that one can form a tuple from normal terms and obtain a normal term. These terms are called *normal tuples* and always have a type of the form $\alpha_1 \times \cdots \times \alpha_n$.

The next result is the *principle of induction on the structure of normal terms.*

Proposition 3.2.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{N}$ satisfying the following conditions.*

1. *If C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \in \mathbb{N})$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{X}$.*
2. *If $t_1, \ldots, t_n \in \mathfrak{X}$, $s_1, \ldots, s_n \in \mathfrak{X}$ $(n \in \mathbb{N})$, $s_0 \in \mathfrak{D}$ and*

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

then

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{X}.$$

3. *If $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \in \mathbb{N})$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{N}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a normal term, it follows that $\mathfrak{N} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{N}$. □

Proposition 3.2.2. $\mathfrak{D} \subseteq \mathfrak{N}$.

Proof. This is a straightforward induction argument on the structure of default terms. □

Proposition 3.2.3. *Each normal term is closed.*

Proof. The proof is by induction on the structure of normal terms. Let $\mathfrak{X}$ be the set of all normal terms that are closed.

Suppose that C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{X}$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$. Then $C\ t_1 \ldots t_n \in \mathfrak{N}$ and $C\ t_1 \ldots t_n$ is closed. Thus $C\ t_1 \ldots t_n \in \mathfrak{X}$.

Suppose next that $t_1, \ldots, t_n \in \mathfrak{X}$, $s_1, \ldots, s_n \in \mathfrak{X}$, and $s_0 \in \mathfrak{D}$ such that $t \equiv \lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L}$. Thus $t \in \mathfrak{N}$ and t is closed, since s_0 is closed by Proposition 3.1.2. Thus $t \in \mathfrak{X}$.

Finally, suppose that $t_1, \ldots, t_n \in \mathfrak{X}$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$. Then $(t_1, \ldots, t_n) \in \mathfrak{N}$ and $(t_1, \ldots, t_n)$ is closed. Thus $(t_1, \ldots, t_n) \in \mathfrak{X}$.

Hence $\mathfrak{X} = \mathfrak{N}$, by Proposition 3.2.1. □

One can gather together all normal terms that have a type more general than some specific closed type.

Definition 3.2.2. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{N}_\alpha = \{t \in \mathfrak{N} \mid t$ has type more general than $\alpha\}$.

The intuitive meaning of $\mathfrak{N}_\alpha$ is that it is the set of terms representing individuals of type α. Note that $\mathfrak{N} = \bigcup_{\alpha \in \mathfrak{S}^c} \mathfrak{N}_\alpha$. However, the $\mathfrak{N}_\alpha$ may not be disjoint. For example, if the alphabet includes *List*, then $[] \in \mathfrak{N}_{List\ \alpha}$, for each closed type α. Furthermore, $\mathfrak{N}_\alpha$ may be empty, for some α.

Example 3.2.1. Assume the alphabet contains just the nullary type constructors M and N (in addition to *1* and Ω) and the data constructors $F : M \to N$ and $G : N \to M$. Then there are no *closed* terms of type M and hence $\mathfrak{N}_M$ is empty.

Since $\mathfrak{D} \subseteq \mathfrak{N}$, it follows that $\mathfrak{D}_\alpha \subseteq \mathfrak{N}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.

Proposition 3.2.4. *If each default data constructor is full, then* $\mathfrak{N}_\alpha \neq \emptyset$*, for each* $\alpha \in \mathfrak{S}^c$*.*

Proof. By Proposition 3.1.4, since each default data constructor is full, $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$. But $\mathfrak{D}_\alpha \subseteq \mathfrak{N}_\alpha$, for each $\alpha \in \mathfrak{S}^c$. Hence the result. □

Proposition 3.2.5.

1. *If* $C\ t_1 \ldots t_n \in \mathfrak{N}_{T\ \alpha_1 \ldots \alpha_k}$*, where* C *has signature* $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ *and* $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$*, then* $t_i \in \mathfrak{N}_{\sigma_i \xi}$*, for* $i = 1, \ldots, n$*.*
2. *If* $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}_{\beta \to \gamma}$*, then* $t_i \in \mathfrak{N}_\beta$ *and* $s_i \in \mathfrak{N}_\gamma$*, for* $i = 1, \ldots, n$*, and* $s_0 \in \mathfrak{D}_\gamma$*.*
3. *If* $(t_1, \ldots, t_n) \in \mathfrak{N}_{\alpha_1 \times \cdots \times \alpha_n}$*, then* $t_i \in \mathfrak{N}_{\alpha_i}$*, for* $i = 1, \ldots, n$*.*

Proof. Suppose first that $C\ t_1 \ldots t_n \in \mathfrak{N}_{T\ \alpha_1 \ldots \alpha_k}$. Then $C\ t_1 \ldots t_n \in \mathfrak{N}$ and so $t_1, \ldots, t_n \in \mathfrak{N}$ (since $\mathfrak{N}$ is the *smallest* set of terms satisfying the conditions in the definition of a normal term). Furthermore, t_i has type ϱ_i, where $\sigma_i \xi = \varrho_i \gamma$, for some γ, for $i = 1, \ldots, n$. Thus $t_i \in \mathfrak{N}_{\sigma_i \xi}$, for $i = 1, \ldots, n$.

Next suppose that $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}_{\beta \to \gamma}$. Hence $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}$ and so $t_i \in \mathfrak{N}$ and $s_i \in \mathfrak{N}$, for $i = 1, \ldots, n$, and $s_0 \in \mathfrak{D}$. Now each t_i has

type more general than β and each s_i has type more general than γ. Hence $t_i \in \mathfrak{N}_\beta$ and $s_i \in \mathfrak{N}_\gamma$, for $i = 1, \ldots, n$, and $s_0 \in \mathfrak{D}_\gamma$.

Finally, suppose that $(t_1, \ldots, t_n) \in \mathfrak{N}_{\alpha_1 \times \cdots \times \alpha_n}$. Then $(t_1, \ldots, t_n) \in \mathfrak{N}$ and so $t_i \in \mathfrak{N}$, for $i = 1, \ldots, n$. Furthermore, each t_i has type more general than α_i. Thus $t_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \ldots, n$. □

Proposition 3.2.6. *If $s, t \in \mathfrak{N}_{\beta \to \gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$, then s and t have the same default value.*

Proof. Let s_0 be the default value for s and t_0 the default value for t. By Proposition 3.2.5, $s_0, t_0 \in \mathfrak{N}_\gamma$. Hence, by Proposition 3.1.5, $s_0 = t_0$. □

Next, a bottom-up characterisation of $\mathfrak{N}$ is provided.

Definition 3.2.3. Define $\{\mathfrak{N}_m\}_{m \in \mathbb{N}}$ inductively as follows.

$$\begin{aligned}
\mathfrak{N}_0 &= \{C \mid C \text{ is a nullary data constructor}\} \\
\mathfrak{N}_{m+1} &= \{C\ t_1 \ldots t_n \in \mathfrak{L} \mid C \text{ is a data constructor of arity } n, \\
&\qquad\qquad t_i \in \mathfrak{N}_m\ (1 \leq i \leq n), n \in \mathbb{N}\} \\
&\quad \cup \{\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L} \mid \\
&\qquad\qquad t_i \in \mathfrak{N}_m, s_i \in \mathfrak{N}_m,\ (1 \leq i \leq n), s_0 \in \mathfrak{D}, n \in \mathbb{N}\} \\
&\quad \cup \{(t_1, \ldots, t_n) \in \mathfrak{L} \mid t_i \in \mathfrak{N}_m\ (1 \leq i \leq n), n \in \mathbb{N}\}.
\end{aligned}$$

Proposition 3.2.7.

1. *$\mathfrak{N}_m \subseteq \mathfrak{N}_{m+1}$, for $m \in \mathbb{N}$.*
2. *$\mathfrak{N} = \bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$.*

Proof. 1. This part is an easy induction argument.

2. First, I show that $\bigcup_{m \in \mathbb{N}} \mathfrak{N}_m \subseteq \mathfrak{N}$. To prove this, it suffices to show by induction that $\mathfrak{N}_m \subseteq \mathfrak{N}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{N}_0 \subseteq \mathfrak{N}$. Suppose next that $\mathfrak{N}_m \subseteq \mathfrak{N}$. It then follows from the definitions of $\mathfrak{N}_{m+1}$ and $\mathfrak{N}$ that $\mathfrak{N}_{m+1} \subseteq \mathfrak{N}$.

Now I show that $\mathfrak{N} \subseteq \bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$. For this, it suffices to show that $\bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{N}$ (since $\mathfrak{N}$ is the smallest such set). Suppose that C is a data constructor, $t_1, \ldots, t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$, and $C\ t_1 \ldots t_n \in \mathfrak{L}$. Since the $\mathfrak{N}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \ldots, t_n \in \mathfrak{N}_p$. Hence $C\ t_1 \ldots t_n \in \mathfrak{N}_{p+1}$ and so $C\ t_1 \ldots t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$. Similar arguments show that $\bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$ satisfies Conditions 2 and 3. □

Proposition 3.2.8. *If the set of data constructors is countable, then $\mathfrak{N}$ is countable.*

Proof. Since the set of data constructors is countable, the set $\mathfrak{T}$ of type constructors is also countable. Thus $\mathfrak{D}$ is countable, by Proposition 3.1.7. Now, since $\mathfrak{N} = \bigcup_{m \in \mathbb{N}} \mathfrak{N}_m$, it suffices to show by induction that $\mathfrak{N}_m$ is countable, for $m \in \mathbb{N}$. However, this follows easily from the definition of $\{\mathfrak{N}_m\}_{m \in \mathbb{N}}$. □

3.3 An Equivalence Relation on Normal Terms

Several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. For example,

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$
$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

and

$$\lambda x.\mathit{if}\ x = 3\ \mathit{then}\ \bot\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

all represent the set $\{1, 2\}$. To reflect this, a relation $\equiv$ is defined on $\mathfrak{N}$.

Definition 3.3.1. The binary relation $\equiv$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s \equiv t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.
2. $\alpha = \beta \to \gamma$, for some β, γ, and
s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$,
t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$
and, $\forall r \in \mathfrak{N}_\beta$,
$(\exists i, j.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge\ s_i \equiv v_j)$
$\vee\ (\exists i.\ r \equiv t_i\ \wedge\ r \not\equiv t_k (\forall k < i)\ \wedge\ r \not\equiv u_j (\forall j)\ \wedge\ s_i \equiv s_0)$
$\vee\ (\exists j.\ r \not\equiv t_i (\forall i)\ \wedge\ r \equiv u_j\ \wedge\ r \not\equiv u_m (\forall m < j)\ \wedge\ s_0 \equiv v_j)$
$\vee\ (r \not\equiv t_i (\forall i)\ \wedge\ r \not\equiv u_j (\forall j)).$
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.

One might conjecture that $\equiv$ is an equivalence relation on $\mathfrak{N}$, but this fails.

Example 3.3.1. Let $A_1 : T_1$ and $A_2 : T_2$ be nullary data constructors, r be $\lambda x.\mathit{if}\ x = A_1\ \mathit{then}\ \bot\ \mathit{else}\ \bot$, s be $\lambda x.\bot$, and t be $\lambda x.\mathit{if}\ x = A_2\ \mathit{then}\ \bot\ \mathit{else}\ \bot$. Then $r, s \in \mathfrak{N}_{T_1 \to \Omega}$ and $r \equiv s$. Furthermore, $s, t \in \mathfrak{N}_{T_2 \to \Omega}$ and $s \equiv t$. But $r \not\equiv t$, since there does not exist $\alpha \in \mathfrak{S}^c$ such that $r, t \in \mathfrak{N}_\alpha$. Thus $\equiv$ is not transitive on $\mathfrak{N}$.

However, all that is really needed is that $\equiv$ be an equivalence relation on each $\mathfrak{N}_\alpha$, and this is indeed true.

Proposition 3.3.1. *For each $\alpha \in \mathfrak{S}^c$, $\equiv\!|_{\mathfrak{N}_\alpha}$ is an equivalence relation on $\mathfrak{N}_\alpha$.*

Proof. It is clear that $\equiv$ is reflexive and symmetric on each $\mathfrak{N}_\alpha$. For transitivity, the proof is by induction on the structure of terms in $\mathfrak{N}$. Suppose $r, s, t \in \mathfrak{N}_\alpha$, $r \equiv s$ and $s \equiv t$. Then there are three cases to consider.

Suppose that $\alpha = T\ \alpha_1 \dots \alpha_k$. Then r is $C\ r_1 \dots r_n$, s is $C\ s_1 \dots s_n$, and t is $C\ t_1 \dots t_n$. Also $r_i \equiv s_i$ and $s_i \equiv t_i$, for $i = 1, \dots, n$. By Proposition 3.2.5 and the induction hypothesis, $r_i \equiv t_i$, for $i = 1, \dots, n$. Hence $r \equiv t$.

Next suppose that $\alpha = \beta \to \gamma$. Then
r is $\lambda z.\mathit{if}\ z = h_1\ \mathit{then}\ k_1\ \mathit{else}\ \dots\ \mathit{if}\ z = h_l\ \mathit{then}\ k_l\ \mathit{else}\ s_0$,
s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$,
and t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \dots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$.
Furthermore, $\forall b \in \mathfrak{N}_\beta$,
$(\exists i, j.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \equiv t_j\ \wedge\ b \not\equiv t_m(\forall m < j)\ \wedge\ k_i \equiv s_j)$
$\vee\ (\exists i.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \not\equiv t_j(\forall j)\ \wedge\ k_i \equiv s_0)$
$\vee\ (\exists j.\ b \not\equiv h_i(\forall i)\ \wedge\ b \equiv t_j\ \wedge\ b \not\equiv t_m(\forall m < j)\ \wedge\ s_0 \equiv s_j)$
$\vee\ (b \not\equiv h_i(\forall i)\ \wedge\ b \not\equiv t_j(\forall j))$
and
$(\exists i, j.\ b \equiv t_i\ \wedge\ b \not\equiv t_k(\forall k < i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_i \equiv v_j)$
$\vee\ (\exists i.\ b \equiv t_i\ \wedge\ b \not\equiv t_k(\forall k < i)\ \wedge\ b \not\equiv u_j(\forall j)\ \wedge\ s_i \equiv s_0)$
$\vee\ (\exists j.\ b \not\equiv t_i(\forall i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_0 \equiv v_j)$
$\vee\ (b \not\equiv t_i(\forall i)\ \wedge\ b \not\equiv u_j(\forall j))$.
Thus, $\forall b \in \mathfrak{N}_\beta$,
$(\exists i, j.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ k_i \equiv v_j)$
$\vee\ (\exists i.\ b \equiv h_i\ \wedge\ b \not\equiv h_k(\forall k < i)\ \wedge\ b \not\equiv u_j(\forall j)\ \wedge\ k_i \equiv s_0)$
$\vee\ (\exists j.\ b \not\equiv h_i(\forall i)\ \wedge\ b \equiv u_j\ \wedge\ b \not\equiv u_m(\forall m < j)\ \wedge\ s_0 \equiv v_j)$
$\vee\ (b \not\equiv h_i(\forall i)\ \wedge\ b \not\equiv u_j(\forall j))$,
by Proposition 3.2.5 and the induction hypothesis. Hence $r \equiv t$.

Finally, suppose that $\alpha = \alpha_1 \times \dots \times \alpha_n$. Then r is $(r_1, \dots, r_n)$, s is $(s_1, \dots, s_n)$ and t is $(t_1, \dots, t_n)$. Also $r_i \equiv s_i$ and $s_i \equiv t_i$, for $i = 1, \dots, n$. By Proposition 3.2.5 and the induction hypothesis, $r_i \equiv t_i$, for $i = 1, \dots, n$. Hence $r \equiv t$. □

Next a neater formulation of the definition of $\equiv$ is provided. This uses the following concept.

Definition 3.3.2. Let t be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{N}_{\beta \to \gamma}$ and $r \in \mathfrak{N}_\beta$. Then $V(t\ r)$ is defined by

$$V(t\ r) = \begin{cases} s_i & \text{if } r \equiv t_i \text{ and } r \not\equiv t_k(\forall k < i) \\ s_0 & \text{if } r \not\equiv t_i(\forall i). \end{cases}$$

Intuitively, $V(t\ r)$ is the 'value' returned when t is applied to r.

Proposition 3.3.2. *Let $t \in \mathfrak{N}_{\beta \to \gamma}$ and $r \in \mathfrak{N}_\beta$. Then $V(t\ r) \in \mathfrak{N}_\gamma$.*

Proof. Let t be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Then $V(t\ r)$ is some s_i, where $i = 0, \dots, n$. By Proposition 3.2.5, each $s_i \in \mathfrak{N}_\beta$. Hence $V(t\ r) \in \mathfrak{N}_\beta$. □

Proposition 3.3.3. *For each $s, t \in \mathfrak{N}$, $s \equiv t$ iff there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.*

1. *$\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $C\ t_1 \ldots t_n$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.*
2. *$\alpha = \beta \rightarrow \gamma$, for some β, γ, and, for all $r \in \mathfrak{N}_\beta$, $V(s\ r) \equiv V(t\ r)$.*
3. *$\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$.*

Proof. The result follows immediately from the definitions of $\equiv$ and $V(t\ r)$. □

3.4 A Total Order on Normal Terms

The equivalence relation $\equiv$ was introduced because several syntactically distinct terms in $\mathfrak{N}$ can represent the same individual. Rather than deal with all the normal terms in an equivalence class in some $\mathfrak{N}_\alpha$, it is preferable to deal with a single representative from the equivalence class. For this purpose, a strict total order on normal terms is introduced. (See the appendix for the definition of a strict total order.)

In the definition of the binary relation $<$ below, it is assumed that, for each $T \in \mathfrak{T}$, there is defined a strict total order $\prec_T$ on the set of all data constructors associated with the type constructor T. For standard types, such as *Int* and *Float*, the usual order provides an appropriate total order. To simplify the statement of the definition, the concept of the trace of an abstraction will be useful.

Definition 3.4.1. Suppose that s is a normal abstraction $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. The *trace* of s, *trace*(s), is the sequence $t_1\, s_1\, t_2\, s_2 \ldots t_n\, s_n$. (For the normal abstraction $\lambda x.s_0$, the trace is the empty sequence ε.)

Definition 3.4.2. The binary relation $<$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s < t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$ and either $C \prec_T D$ or $C = D$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.
2. $\alpha = \beta \rightarrow \gamma$, for some β, γ, and *trace*$(s) <$ *trace*(t), where $<$ is the induced lexicographic ordering.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $1 \leq j \leq n$, $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$ and $s_j < t_j$.

Note 3.4.1. Before continuing, a closer examination of what it means for two normal terms to be identical is needed. Consider the two terms

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \text{ and}$$
$$\lambda y.\mathit{if}\ y = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0.$$

The only difference between these two terms is that the bound variables in each of them have distinct names. In fact, this difference is quite inessential (and could be removed altogether with a suitable notation) and hence I prefer to regard the terms as being identical. Generally, identity of normal terms is regarded as meaning α-equivalent, that is, the names of bound variables are not important. This convention is used implicitly in the remainder of the book.

Proposition 3.4.1. *For each $\alpha \in \mathfrak{S}^c$, $<|_{\mathfrak{N}_\alpha}$ is a strict total order on $\mathfrak{N}_\alpha$.*

Proof. The proof is by induction on the structure of terms in $\mathfrak{N}$.
First, I show that $<$ is irreflexive. Let $t \in \mathfrak{N}_\alpha$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then t is $C\ t_1 \ldots t_n$. Thus $t \not< t$, since $t_i \not< t_i$, for $1 \leq i \leq n$, by the induction hypothesis.
2. Let $\alpha = \beta \to \gamma$. Then t is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Thus $t \not< t$, since $t_i \not< t_i$ and $s_i \not< s_i$, for $1 \leq i \leq n$, by the induction hypothesis.
3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then t is $(t_1, \ldots, t_n)$. Thus $t \not< t$, since $t_i \not< t_i$, for $1 \leq i \leq n$, by the induction hypothesis.

Next, I show that $<$ is asymmetric. Let $s, t \in \mathfrak{N}_\alpha$ and suppose that $s < t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$. If $C \prec_T D$, then $D \not\prec_T C$ and $D \neq C$ and so $t \not< s$. Otherwise, $C = D$, in which case there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j < t_j$ and $1 \leq j \leq n$. Hence $t_1 \not< s_1, \ldots, t_{j-1} \not< s_{j-1}$, and $t_j \neq s_j$, since $<$ is irreflexive. By the induction hypothesis, $t_j \not< s_j$. Thus $t \not< s$.
2. Let $\alpha = \beta \to \gamma$. Then $\mathit{trace}(s) < \mathit{trace}(t)$. By examining each of the two cases in the definition of the induced lexicographic ordering and using the induction hypothesis, one can see that $\mathit{trace}(t) \not< \mathit{trace}(s)$. Thus $t \not< s$.
3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j < t_j$ and $1 \leq j \leq n$. Hence $t_1 \not< s_1, \ldots, t_{j-1} \not< s_{j-1}$, and $t_j \neq s_j$ since $<$ is irreflexive. By the induction hypothesis, $t_j \not< s_j$. Thus $t \not< s$.

Next, I show that $<$ is transitive. Let $r, s, t \in \mathfrak{N}_\alpha$ and suppose that $r < s$ and $s < t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $B\ r_1 \ldots r_p$, s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$, either $B \prec_T C$ or $B = C$ and there exists j such that $r_1 = s_1, \ldots, r_{j-1} = s_{j-1}$, $r_j < s_j$ and $1 \leq j \leq n$, and either $C \prec_T D$ or $C = D$ and there

exists k such that $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, $s_k < t_k$ and $1 \leq k \leq n$. If $B = C = D$, by the induction hypothesis, there exists l such that $r_1 = t_1, \ldots, r_{l-1} = t_{l-1}$, $r_l < t_l$ and $1 \leq l \leq n$. Hence $r < t$. If either $B \prec_T C$ or $C \prec_T D$, then $B \prec_T D$. Thus $r < t$.
2. Let $\alpha = \beta \rightarrow \gamma$. Then $trace(r) < trace(s)$ and $trace(s) < trace(t)$. By examining each of the two cases in the definition of the induced lexicographic ordering and using the induction hypothesis, one can see that $trace(r) < trace(t)$. Thus $r < t$.
3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$ t is $(t_1, \ldots, t_n)$, there exists j such that $r_1 = s_1, \ldots, r_{j-1} = s_{j-1}$, $r_j < s_j$ and $1 \leq j \leq n$, and there exists k such that $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, $s_k < t_k$ and $1 \leq k \leq n$. Then, by the induction hypothesis, there exists p such that $r_1 = t_1, \ldots, r_{p-1} = t_{p-1}$, $r_p < t_p$ and $1 \leq p \leq n$. Thus $r < t$.

Finally, I show that $<$ is total. Let $s, t \in \mathfrak{N}_\alpha$ and suppose $s \neq t$.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$. If $C \neq D$, then either $C \prec_T D$ or $D \prec_T C$, since $\prec_T$ is total. Thus either $s < t$ or $t < s$. Otherwise, $C = D$, in which case there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j \neq t_j$ and $1 \leq j \leq n$. By the induction hypothesis, either $s_j < t_j$ or $t_j < s_j$. Thus either $s < t$ or $t < s$.
2. Let $\alpha = \beta \rightarrow \gamma$. Then $trace(s) \neq trace(t)$. If $trace(s)$ is a (proper) prefix of $trace(t)$, then $s < t$. If $trace(t)$ is a (proper) prefix of $trace(s)$, then $t < s$. Otherwise, there exists an index at which $trace(s)$ and $trace(t)$ differ. Using the induction hypothesis, one can see that either $trace(s) < trace(t)$ or $trace(t) < trace(s)$. Thus either $s < t$ or $t < s$.
3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 = t_1, \ldots, s_{j-1} = t_{j-1}$, $s_j \neq t_j$ and $1 \leq j \leq n$. By the induction hypothesis, either $s_j < t_j$ or $t_j < s_j$. Thus either $s < t$ or $t < s$. □

3.5 Basic Terms

The definition of the key concept of a basic term can now be given.

Definition 3.5.1. The set of *basic terms*, $\mathfrak{B}$, is defined inductively as follows.

1. If C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{B}$ ($n \in \mathbb{N}$) such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{B}$.
2. If $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}$, $t_1 < \cdots < t_n$, $s_i \notin \mathfrak{D}$, for $1 \leq i \leq n$ ($n \in \mathbb{N}$), $s_0 \in \mathfrak{D}$ and

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L},$$

then

$$\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{B}.$$

3. If $t_1, \ldots, t_n \in \mathfrak{B}$ $(n \in \mathbb{N})$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{B}$.

The basic terms from Part 1 of the definition are called *basic structures*, those from Part 2 are called *basic abstractions*, and those from Part 3 are called *basic tuples.*

Note 3.5.1. A suitable universe for the construction of Definition 3.5.1 is the set $\mathfrak{N}$ of all normal terms. Thus the restriction $t_1 < \cdots < t_n$ assumed on $t_1, \ldots, t_n$ in Condition 2 is well-defined.

The next result is the *principle of induction on the structure of basic terms.*

Proposition 3.5.1. *Let $\mathfrak{X}$ be a subset of $\mathfrak{B}$ satisfying the following conditions.*

1. *If C is a data constructor of arity n and $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \in \mathbb{N})$ such that $C\ t_1 \ldots t_n \in \mathfrak{L}$, then $C\ t_1 \ldots t_n \in \mathfrak{X}$.*
2. *If $t_1, \ldots, t_n \in \mathfrak{X}$, $s_1, \ldots, s_n \in \mathfrak{X}$, $t_1 < \cdots < t_n$, $s_i \notin \mathfrak{D}$, for $1 \leq i \leq n$ $(n \in \mathbb{N})$, $s_0 \in \mathfrak{D}$ and*

$$\lambda x.\textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0 \in \mathfrak{L},$$

   then

$$\lambda x.\textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0 \in \mathfrak{X}.$$

3. *If $t_1, \ldots, t_n \in \mathfrak{X}$ $(n \in \mathbb{N})$ and $(t_1, \ldots, t_n) \in \mathfrak{L}$, then $(t_1, \ldots, t_n) \in \mathfrak{X}$.*

Then $\mathfrak{X} = \mathfrak{B}$.

Proof. Since $\mathfrak{X}$ satisfies Conditions 1 to 3 of the definition of a basic term, it follows that $\mathfrak{B} \subseteq \mathfrak{X}$. Thus $\mathfrak{X} = \mathfrak{B}$. □

Proposition 3.5.2. $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$.

Proof. The first of these inclusions is an easy induction argument on the structure of default terms, while the second is a similar induction argument on the structure of basic terms. □

Since $\mathfrak{B} \subseteq \mathfrak{N}$, it follows from Proposition 3.2.3 that each basic term is closed.

Proposition 3.5.3. *If the set of data constructors is countable, then $\mathfrak{B}$ is countable.*

Proof. By Proposition 3.2.8, $\mathfrak{N}$ is countable. By Proposition 3.5.2, $\mathfrak{B} \subseteq \mathfrak{N}$. □

As for normal terms, the basic terms of a particular type can be gathered together.

Definition 3.5.2. For each $\alpha \in \mathfrak{S}^c$, define $\mathfrak{B}_\alpha = \{t \in \mathfrak{B} \mid t$ has type more general than $\alpha\}$.

The sets $\{\mathfrak{B}_\alpha\}_{\alpha \in \mathfrak{S}^c}$ play an important role in knowledge representation. For example, for a particular machine learning application, the representation space of the individuals would be $\mathfrak{B}_\alpha$, for some choice of $\alpha \in \mathfrak{S}^c$.

Example 3.5.1. If the numerical data constructors are identified with the corresponding numbers, then $\mathfrak{B}_{Float}$ is $\mathbb{F}$, $\mathfrak{B}_{Nat}$ is $\mathbb{N}$, and $\mathfrak{B}_{Int}$ is $\mathbb{Z}$. Also $\mathfrak{B}_{Nat \to \Omega}$ is (isomorphic to) the set of all finite subsets of $\mathbb{N}$ (assuming that $\bot$ is the default data constructor for Ω). Finally, $\mathfrak{B}_{Nat \times \cdots \times Nat}$ is $\mathbb{N}^n$, where there are n components in the product type.

Proposition 3.5.4. *For each $\alpha \in \mathfrak{S}^c$, $\mathfrak{D}_\alpha \subseteq \mathfrak{B}_\alpha \subseteq \mathfrak{N}_\alpha$.*

Proof. The result follows directly from the fact that $\mathfrak{D} \subseteq \mathfrak{B} \subseteq \mathfrak{N}$. □

Proposition 3.5.5.

1. *If $C\ t_1 \dots t_n \in \mathfrak{B}_{T\ \alpha_1 \dots \alpha_k}$, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \dots a_k)$ and $\xi = \{a_1/\alpha_1, \dots, a_k/\alpha_k\}$, then $t_i \in \mathfrak{B}_{\sigma_i \xi}$, for $i = 1, \dots, n$.*
2. *If $\lambda x.if\ x = t_1\ then\ s_1\ else\ \dots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{B}_{\beta \to \gamma}$, then $t_i \in \mathfrak{B}_\beta$ and $s_i \in \mathfrak{B}_\gamma$, for $i = 1, \dots, n$, and $s_0 \in \mathfrak{D}_\gamma$.*
3. *If $(t_1, \dots, t_n) \in \mathfrak{B}_{\alpha_1 \times \cdots \times \alpha_n}$, then $t_i \in \mathfrak{B}_{\alpha_i}$, for $i = 1, \dots, n$.*

Proof. Suppose first that $C\ t_1 \dots t_n \in \mathfrak{B}_{T\ \alpha_1 \dots \alpha_k}$. Then $C\ t_1 \dots t_n \in \mathfrak{B}$ and so $t_1, \dots, t_n \in \mathfrak{B}$ (since $\mathfrak{B}$ is the *smallest* set of terms satisfying the conditions in the definition of a basic term). Furthermore, t_i has type ϱ_i, where $\sigma_i \xi = \varrho_i \gamma$, for some γ, for $i = 1, \dots, n$. Thus $t_i \in \mathfrak{B}_{\sigma_i \xi}$, for $i = 1, \dots, n$.

Next suppose that $\lambda x.if\ x = t_1\ then\ s_1\ else\ \dots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{B}_{\beta \to \gamma}$. Hence $\lambda x.if\ x = t_1\ then\ s_1\ else\ \dots\ if\ x = t_n\ then\ s_n\ else\ s_0 \in \mathfrak{B}$ and so $t_i \in \mathfrak{B}$ and $s_i \in \mathfrak{B}$, for $i = 1, \dots, n$, and $s_0 \in \mathfrak{D}$. Now each t_i has type more general than β and each s_i has type more general than γ. Hence $t_i \in \mathfrak{B}_\beta$ and $s_i \in \mathfrak{B}_\gamma$, for $i = 1, \dots, n$, and $s_0 \in \mathfrak{D}_\gamma$.

Finally, suppose that $(t_1, \dots, t_n) \in \mathfrak{B}_{\alpha_1 \times \cdots \times \alpha_n}$. Then $(t_1, \dots, t_n) \in \mathfrak{B}$ and so $t_i \in \mathfrak{B}$, for $i = 1, \dots, n$. Furthermore, each t_i has type more general than α_i. Thus $t_i \in \mathfrak{B}_{\alpha_i}$, for $i = 1, \dots, n$. □

Proposition 3.5.6. *Let $t \in \mathfrak{B}_{\beta \to \gamma}$ and $r \in \mathfrak{B}_\beta$. Then $V(t\ r) \in \mathfrak{B}_\gamma$.*

Proof. Let t be $\lambda x.if\ x = t_1\ then\ s_1\ else\ \dots\ if\ x = t_n\ then\ s_n\ else\ s_0$. Then $V(t\ r)$ is some s_i, where $i = 0, \dots, n$. By Proposition 3.5.5, each $s_i \in \mathfrak{B}_\beta$. Hence $V(t\ r) \in \mathfrak{B}_\beta$. □

Proposition 3.5.7. *If each default data constructor is full, then $\mathfrak{B}_\alpha \neq \emptyset$, for all $\alpha \in \mathfrak{S}^c$.*

Proof. By Proposition 3.1.4, since each default data constructor is full, $\mathfrak{D}_\alpha \neq \emptyset$, for each $\alpha \in \mathfrak{S}^c$. But $\mathfrak{D}_\alpha \subseteq \mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$. Hence the result. □

Next, a bottom-up characterisation of $\mathfrak{B}$ is provided.

Definition 3.5.3. Define $\{\mathfrak{B}_m\}_{m \in \mathbb{N}}$ inductively as follows.

$$\begin{aligned}
\mathfrak{B}_0 &= \{C \mid C \text{ is a nullary data constructor}\} \\
\mathfrak{B}_{m+1} &= \{C\ t_1 \dots t_n \in \mathfrak{L} \mid C \text{ is a data constructor of arity } n, \\
&\qquad\qquad t_i \in \mathfrak{B}_m\ (1 \leq i \leq n), n \in \mathbb{N}\} \\
&\quad \cup \{\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \dots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0 \in \mathfrak{L} \mid \\
&\qquad\qquad t_i \in \mathfrak{B}_m, s_i \in \mathfrak{B}_m, s_i \notin \mathfrak{D}\ (1 \leq i \leq n), \\
&\qquad\qquad t_1 < \dots < t_n, s_0 \in \mathfrak{D}, n \in \mathbb{N}\} \\
&\quad \cup \{(t_1, \dots, t_n) \in \mathfrak{L} \mid t_i \in \mathfrak{B}_m\ (1 \leq i \leq n), n \in \mathbb{N}\}.
\end{aligned}$$

Proposition 3.5.8.

1. $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, *for* $m \in \mathbb{N}$.
2. $\mathfrak{B} = \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$.

Proof. 1. This part is an easy induction argument.

2. First, I show that $\bigcup_{m \in \mathbb{N}} \mathfrak{B}_m \subseteq \mathfrak{B}$. To prove this, it suffices to show by induction that $\mathfrak{B}_m \subseteq \mathfrak{B}$, for $m \in \mathbb{N}$. Clearly $\mathfrak{B}_0 \subseteq \mathfrak{B}$. Suppose next that $\mathfrak{B}_m \subseteq \mathfrak{B}$. It then follows from the definitions of $\mathfrak{B}_{m+1}$ and $\mathfrak{B}$ that $\mathfrak{B}_{m+1} \subseteq \mathfrak{B}$.

Now I show that $\mathfrak{B} \subseteq \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$. For this, it suffices to show that $\bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$ satisfies Conditions 1, 2 and 3 in the definition of $\mathfrak{B}$ (since $\mathfrak{B}$ is the smallest such set). Suppose that C is a data constructor, $t_1, \dots, t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$, and $C\ t_1 \dots t_n \in \mathfrak{L}$. Since the $\mathfrak{B}_m$ are increasing, there exists $p \in \mathbb{N}$ such that $t_1, \dots, t_n \in \mathfrak{B}_p$. Hence $C\ t_1 \dots t_n \in \mathfrak{B}_{p+1}$ and so $C\ t_1 \dots t_n \in \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$. Similar arguments show that $\bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$ satisfies Conditions 2 and 3. □

The next result shows that, for basic terms, the equivalence relation $\equiv$ reduces to the identity relation (up to α-equivalence, of course).

Proposition 3.5.9. *Let* $s, t \in \mathfrak{B}$. *Then* $s \equiv t$ *iff* $s = t$.

Proof. If $s = t$, then it is clear that $s \equiv t$. Suppose now that $s \equiv t$. By Proposition 3.3.3, there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$ with three cases to consider.

1. $\alpha = T\ \alpha_1 \dots \alpha_k$, for some $T, \alpha_1, \dots, \alpha_k$, and s is $C\ s_1 \dots s_n$, t is $C\ t_1 \dots t_n$ and $s_i \equiv t_i$, for $i = 1, \dots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \dots, n$. Thus $s = t$.

2. $\alpha = \beta \to \gamma$, for some β, γ, and, for all $r \in \mathfrak{B}_\beta$, $V(s\ r) \equiv V(t\ r)$. Suppose that
   s is $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$, and
   t is $\lambda y.\mathit{if}\ y = u_1\ \mathit{then}\ v_1\ \mathit{else}\ \ldots\ \mathit{if}\ y = u_m\ \mathit{then}\ v_m\ \mathit{else}\ s_0$.
   Then $V(s\ t_i) \equiv V(t\ t_i)$, for $i = 1, \ldots, n$, and $V(s\ u_j) \equiv V(t\ u_j)$, for $j = 1, \ldots, m$. By the induction hypothesis, $V(s\ t_i) = V(t\ t_i)$, for $i = 1, \ldots, n$, and $V(s\ u_j) = V(t\ u_j)$, for $j = 1, \ldots, m$. That is, $V(t\ t_i) = s_i$, for $i = 1, \ldots, n$, and $V(s\ u_j) = v_j$, for $j = 1, \ldots, m$. It follows that $\{t_1, \ldots, t_n\} = \{u_1, \ldots, u_m\}$. Since each of s and t is basic, one can see that $s = t$.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $s_i \equiv t_i$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$. □

Proposition 3.5.10. *Let $s, t \in \mathfrak{B}_{\beta \to \gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$. Then $s = t$ iff $V(s\ u) = V(t\ u)$, for all $u \in \mathfrak{B}_\beta$.*

Proof. The result follows immediately from Propositions 3.3.3 and 3.5.9. □

Definition 3.5.4. Let $u \in \mathfrak{B}_{\beta \to \gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$. The *support* of u, denoted $\mathit{supp}(u)$, is the set $\{v \in \mathfrak{B}_\beta \mid V(u\ v) \notin \mathfrak{D}\}$.

Proposition 3.5.11. *Let $u \in \mathfrak{B}_{\beta \to \gamma}$, for some $\beta, \gamma \in \mathfrak{S}^c$. Then $\mathit{supp}(u)$ is a finite set.*

Proof. Let u be $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$. Now Proposition 3.5.9 shows that if $s, t \in \mathfrak{B}$, then $s \equiv t$ iff $s = t$. Thus $V(u\ v) \notin \mathfrak{D}$ iff $v \in \{t_1, \ldots, t_n\}$. Hence $\mathit{supp}(u)$ is finite. □

The proof of Proposition 3.5.11 shows that if u is the term $\lambda x.\mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$, then $\mathit{supp}(u) = \{t_1, \ldots, t_n\}$.

The next proposition justifies restricting attention to basic terms for knowledge representation purposes.

Proposition 3.5.12. *If $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, then there is a unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$.*

Proof. Uniqueness follows immediately from Proposition 3.5.9, thus only existence has to be shown. The proof of existence is by induction on the structure of s. There are three cases to consider.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$. Suppose that s is $C\ s_1 \ldots s_n$, where C has signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$. Let $\xi = \{a_1/\alpha_1, \ldots, a_k/\alpha_k\}$. By Proposition 3.2.5, $s_i \in \mathfrak{N}_{\sigma_i \xi}$, for $i = 1, \ldots, n$. By the induction hypothesis, there exist $t_1, \ldots, t_n \in \mathfrak{B}_{\sigma_i \xi}$ such that $s_i \equiv t_i$, for $i = 1, \ldots, n$. Then $C\ t_1 \ldots t_n \in \mathfrak{B}_{T\ \alpha_1 \ldots \alpha_k}$ and, by Proposition 3.3.1, $s \equiv C\ t_1 \ldots t_n$.

2. $\alpha = \beta \to \gamma$, for some β, γ. Suppose that s is

$$\lambda x.\textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \dots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0.$$

By Proposition 3.2.5, $t_i \in \mathfrak{N}_\beta$ and $s_i \in \mathfrak{N}_\gamma$, for $i = 1, \dots, n$. By the induction hypothesis, there exist $t'_i \in \mathfrak{B}_\beta$ and $s'_i \in \mathfrak{B}_\gamma$ such that $t_i \equiv t'_i$ and $s_i \equiv s'_i$, for $i = 1, \dots, n$. Now let s' be

$$\lambda x.\textit{if } x = t'_1 \textit{ then } s'_1 \textit{ else } \dots \textit{ if } x = t'_n \textit{ then } s'_n \textit{ else } s_0.$$

By Proposition 3.3.3, $s \equiv s'$. For any $\{i_1, \dots, i_p\}$ $(p \geq 2)$ such that $i_1 < \dots < i_p$ and $t'_{i_1} = \dots = t'_{i_p}$, drop from s' the components of the if-then-else containing $x = t'_{i_2}, \dots, x = t'_{i_p}$ to obtain s'' of the form

$$\lambda x.\textit{if } x = t''_1 \textit{ then } s''_1 \textit{ else } \dots \textit{ if } x = t''_m \textit{ then } s''_m \textit{ else } s_0.$$

By Proposition 3.3.3, $s \equiv s''$. Also drop any components of the if-then-else in s'' for which the value s''_i is in $\mathfrak{D}$ to obtain s''' of the form

$$\lambda x.\textit{if } x = t'''_1 \textit{ then } s'''_1 \textit{ else } \dots \textit{ if } x = t'''_k \textit{ then } s'''_k \textit{ else } s_0.$$

By Proposition 3.3.3, $s \equiv s'''$. At this stage, all the t'''_i are pairwise distinct and all the s'''_i are not in $\mathfrak{D}$. Finally, let $t''''_1 \dots t''''_k$ be the result of ordering the sequence $t'''_1 \dots t'''_k$ according to the total order on $\mathfrak{N}_\beta$ induced by the $\prec_T$. Let t be

$$\lambda x.\textit{if } x = t''''_1 \textit{ then } s''''_1 \textit{ else } \dots \textit{ if } x = t''''_k \textit{ then } s''''_k \textit{ else } s_0,$$

the result of the corresponding reordering of the components of the if-then-else in s'''. Then $t \in \mathfrak{B}_{\beta \to \gamma}$ and, by Proposition 3.3.3, $s \equiv t$.

3. $\alpha = \alpha_1 \times \dots \times \alpha_n$, for some $\alpha_1, \dots, \alpha_n$. Suppose s is $(s_1, \dots, s_n)$. By Proposition 3.2.5, $s_i \in \mathfrak{N}_{\alpha_i}$, for $i = 1, \dots, n$. By the induction hypothesis, there exist $t_i \in \mathfrak{B}_{\alpha_i}$ such that $s_i \equiv t_i$, for $i = 1, \dots, n$. Then $(t_1, \dots, t_n) \in \mathfrak{B}_{\alpha_1 \times \dots \times \alpha_n}$ and, by Proposition 3.3.3, $s \equiv (t_1, \dots, t_n)$. □

Proposition 3.5.12 justifies the next definition.

Definition 3.5.5. Let $s \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. The unique $t \in \mathfrak{B}_\alpha$ such that $s \equiv t$ is called the *basic form* of s.

The next result provides an alternative characterisation of the equivalence relation $\equiv$.

Proposition 3.5.13. *Let $s, t \in \mathfrak{N}_\alpha$, for some $\alpha \in \mathfrak{S}^c$. Then $s \equiv t$ iff s and t have the same basic form.*

Proof. Suppose that $s \equiv t$. Let s' be the basic form of s and t' the basic form of t. Then, by definition, $s \equiv s'$ and $t \equiv t'$. Since $\equiv$ is an equivalence relation, $s' \equiv t'$. By Proposition 3.5.9, $s' = t'$.

Conversely, suppose that s and t have the same basic form, say, r. Then $s \equiv r$ and $t \equiv r$, and so $s \equiv t$. □

Based on the proof of Proposition 3.5.12, one can give an algorithm that computes the basic form of a normal term. This algorithm, for the function *Reduce*, is given in Fig. 3.1 below.

function $Reduce(s, \{\prec_T\}_{T \in \mathfrak{T}})$ **returns** the basic form of s;
input: s, a normal term;
  $\{\prec_T\}_{T \in \mathfrak{T}}$, a set of total orders, one for each type constructor T;

case s **of**
$C\ s_1 \ldots s_n$:

  for $i = 1, \ldots, n$ **do**
    $t_i := Reduce(s_i, \{\prec_T\}_{T \in \mathfrak{T}})$;
  return $C\ t_1 \ldots t_n$;

$\lambda x.if\ x = t_1\ then\ s_1\ else\ \ldots\ if\ x = t_n\ then\ s_n\ else\ s_0$:
  for $i = 1, \ldots, n$ **do**
    $t_i' := Reduce(t_i, \{\prec_T\}_{T \in \mathfrak{T}})$;
    $s_i' := Reduce(s_i, \{\prec_T\}_{T \in \mathfrak{T}})$;
  $s' := \lambda x.\ if\ x = t_1'\ then\ s_1'\ else\ \ldots\ if\ x = t_n'\ then\ s_n'\ else\ s_0$;
  $s'' := s'$ modified by dropping occurrences of components in s'
    containing duplicate occurrences of some t_i';
  $s''' := s''$ modified by dropping occurrences of components in s''
    whose value is a default term;
  $t := s'''$ modified by reordering components in s''' according to
    the total ordering induced by $\{\prec_T\}_{T \in \mathfrak{T}}$;
  return t;

$(s_1, \ldots, s_n)$:
  for $i = 1, \ldots, n$ **do**
    $t_i := Reduce(s_i, \{\prec_T\}_{T \in \mathfrak{T}})$;
  return $(t_1, \ldots, t_n)$;

Fig. 3.1. Algorithm for computing the basic form of a normal term

Here is an example to show how this algorithm works.

Example 3.5.2. Let s be the normal term

$$\lambda x.if\ x = 3\ then\ \bot\ else\ if\ x = 2\ then\ \top\ else\ if\ x = 1\ then\ \top\ else\ if\ x = 3\ then\ \top\ else\ \bot.$$

Assume that the total order on the integers is the usual order. In the first step, each of $1, 2, 3$, and $\top$ and $\bot$ are replaced by their basic form. Since each of these is already a basic term, this step has no effect. Second, the component of the if-then-else containing the duplicated occurrence of $x = 3$ is dropped to obtain

$$\lambda x.if\ x = 3\ then\ \bot\ else\ if\ x = 2\ then\ \top\ else\ if\ x = 1\ then\ \top\ else\ \bot.$$

Third, the component containing the occurrence $x = 3$ is dropped since the corresponding value is $\bot$ to obtain

$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Finally, the sequence 2 1 is ordered according to the total order and the components of the if-then-else are reordered accordingly to obtain

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot,$$

which is the basic form of s.

Example 3.5.3. As an example of the representation of individuals using basic terms, consider the problem of modelling a chemical molecule. The first issue is to choose a suitable type to represent a molecule. I use an undirected graph to model a molecule – an atom is a vertex in the graph and a bond is an edge. Having made this choice, suitable types are then set up for the atoms and bonds. For this, I first introduce the type *Element*, which is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : \mathit{Element}.$$

This declaration declares the constants *Br* (bromine), *C* (carbon), and so on, to be constants of type *Element*. I also make the following type synonyms.

$$\begin{aligned}
&\mathit{AtomType} = \mathit{Nat}\\
&\mathit{Charge} = \mathit{Float}\\
&\mathit{Atom} = \mathit{Element} \times \mathit{AtomType} \times \mathit{Charge}\\
&\mathit{Bond} = \mathit{Nat}.
\end{aligned}$$

The expression '$\mathit{AtomType} = \mathit{Nat}$', for example, simply states that the type *AtomType* is nothing other than *Nat*. The use of type synonyms greatly improves the readability of declarations.

The type of an (undirected) graph is $\mathit{Graph}\ \nu\ \varepsilon$, where ν is the type of information associated with the vertices and ε is the type of information associated with the edges. *Graph* is defined as follows.

$$\begin{aligned}
&\mathit{Label} = \mathit{Nat}\\
&\mathit{Graph}\ \nu\ \varepsilon = \{\mathit{Label} \times \nu\} \times \{(\mathit{Label} \to \mathit{Nat}) \times \varepsilon\}.
\end{aligned}$$

Here the multisets of type $\mathit{Label} \to \mathit{Nat}$ are intended to all have cardinality 2, that is, they are intended to be regarded as *unordered* pairs. (A multiset is a function mapping into the natural numbers for which the value of the function on some element is its multiplicity, that is, the number of times it occurs in the multiset.) Note that this definition corresponds closely to the mathematical definition of a graph: each vertex is labelled by a unique

natural number and each edge is uniquely labelled by the unordered pair of labels of the vertices that it connects.

I now declare the type of a molecule to be an (undirected) graph whose vertices have type *Atom* and whose edges have type *Bond*. This leads to the following type synonym.

$$Molecule = Graph\ Atom\ Bond.$$

Here is an example molecule, called d_1, from the well-known Mutagenesis dataset. The notation $\langle s, t \rangle$ is used as a shorthand for the multiset that takes the value 1 on each of s and t, and is 0 elsewhere. Thus $\langle s, t \rangle$ is essentially an unordered pair.

$$\begin{aligned}
&(\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(C,22,-0.117)),\\
&\quad(7,(H,3,0.142)),(8,(H,3,0.143)),(9,(H,3,0.142)),\\
&\quad(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\quad(13,(C,22,-0.117)),(14,(C,22,-0.117)),(15,(H,3,0.143)),\\
&\quad(16,(H,3,0.143)),(17,(C,22,-0.117)),(18,(C,22,-0.117)),\\
&\quad(19,(C,22,-0.117)),(20,(C,22,-0.117)),(21,(H,3,0.142)),\\
&\quad(22,(H,3,0.143)),(23,(H,3,0.142)),(24,(N,38,0.812)),\\
&\quad(25,(O,40,-0.388)),(26,(O,40,-0.388))\},\\
&\ \{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&\quad(\langle 5,14\rangle,7),(\langle 6,10\rangle,1),(\langle 11,12\rangle,7),(\langle 11,17\rangle,7),\\
&\quad(\langle 12,13\rangle,7),(\langle 12,20\rangle,7),(\langle 13,14\rangle,7),(\langle 13,15\rangle,1),\\
&\quad(\langle 14,16\rangle,1),(\langle 17,18\rangle,7),(\langle 17,21\rangle,1),(\langle 18,19\rangle,7),\\
&\quad(\langle 18,22\rangle,1),(\langle 19,20\rangle,7),(\langle 19,24\rangle,1),(\langle 20,23\rangle,1),\\
&\quad(\langle 24,25\rangle,2),(\langle 24,26\rangle,2)\}).
\end{aligned}$$

3.6 Metrics on Basic Terms

For a number of reasons, it is important to have a metric defined on basic terms. For example, in instance-based learning, such a metric is needed to determine those terms that are 'nearby' some given term. Thus I give now the definition of a suitable function d from $\mathfrak{B} \times \mathfrak{B}$ into $\mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers.

First, the definition of a metric space is recalled.

Definition 3.6.1. A *pseudometric space* $(\mathcal{X}, d)$ is a (non-empty) set $\mathcal{X}$ and function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ satisfying the following conditions.

1. $d(x, y) \geq 0$, for all $x, y \in \mathfrak{X}$.
2. $d(x, y) = d(y, x)$, for all $x, y \in \mathfrak{X}$.
3. $x = y$ implies $d(x, y) = 0$, for all $x, y \in \mathfrak{X}$.
4. $d(x, z) \leq d(x, y) + d(y, z)$, for all $x, y, z \in \mathfrak{X}$.

The function d is called a *pseudometric*

Definition 3.6.2. A *metric space* is a pseudometric space $(\mathfrak{X}, d)$ such that d satisfies the property that $d(x, y) = 0$ implies $x = y$, for all $x, y \in \mathfrak{X}$. In this case, d is called a *metric.*

The definition of the metric on $\mathfrak{B}$ depends upon some given functions ϱ_T, for $T \in \mathfrak{T}$, and φ. The ϱ_T are assumed to satisfy the following conditions.

1. For each $T \in \mathfrak{T}$, ϱ_T is a metric on the set of data constructors associated with T.
2. If there is at least one data constructor of arity > 0 associated with T, then ϱ_T is the discrete metric.

For example, the type constructor *List* has two data constructors $\sharp$ (of arity > 0) and $[]$, and so $\varrho_{List}([], \sharp) = 1$. In contrast, *Nat* has only nullary data constructors and hence the second condition does not apply. The next example gives typical choices for ϱ_T, for various T.

Example 3.6.1. For the type *1*, ϱ_1 could be the discrete metric. Similarly, for ϱ_Ω. For the type *Nat*, one could use $\varrho_{Nat}(n, m) = |n - m|$. Similarly, for *Int* and *Float*. For a type constructor like *Shape* in Sect. 3.2, it is natural to employ the discrete metric on the set of data constructors $\{Circle, Rectangle\}$.

The second function φ must be a non-decreasing function from the non-negative reals into the closed interval $[0, 1]$ such that $\varphi(0) = 0$, $\varphi(x) > 0$ if $x > 0$, and $\varphi(x + y) \leq \varphi(x) + \varphi(y)$, for each x and y.

Example 3.6.2. Typical choices for φ could be $\varphi(x) = \frac{x}{1+x}$ or $\varphi(x) = \min\{1, x\}$.

Definition 3.6.3. The function $d : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively as follows. Let $s, t \in \mathfrak{B}$.

1. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = T\, \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, then
$$d(s, t) = \begin{cases} \varrho_T(C, D) & \text{if } C \neq D \\ (1/2) \max_{i=1,\ldots,n} \varphi(d(s_i, t_i)) & \text{otherwise} \end{cases}$$
where s is $C\, s_1 \ldots s_n$ and t is $D\, t_1 \ldots t_m$.
2. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then
$$d(s, t) = \sum_{r \in supp(s) \cup supp(t)} d(V(s\, r), V(t\, r)).$$

3. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, then

$$d(s,t) = \sum_{i=1}^{n} d(s_i, t_i),$$

where s is $(s_1, \ldots, s_n)$ and t is $(t_1, \ldots, t_n)$.
4. If there does not exist $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$, then $d(s,t) = 1$.

Definition 3.6.3 is an inductive definition that depends on the fact that $\mathfrak{B} \times \mathfrak{B}$ is well-founded under the order $\prec_2$ given by the product of the substring order $\prec$ (Example A.1.1) on $\mathfrak{B}$ with itself. Thus the principle of inductive construction for well-founded sets (Proposition A.1.4) can be applied. Note that the minimal elements of $\mathfrak{B}$ under $\prec$ are the basic terms that do not have proper subterms, that is, the nullary data constructors. Thus the minimal elements of $\mathfrak{B} \times \mathfrak{B}$ under $\prec_2$ are tuples of the form (C, D), where C and D are nullary data constructors. The well-definedness of the function d depends upon Proposition A.1.4: d is defined directly on the minimal elements (that is, pairs of the form (C, D), where C and D are nullary data constructors) and, for other pairs, is uniquely determined by the rules for each of the three kinds of basic terms and Part 4 when the types do not match.

In Part 1 of the definition, if $n = 0$, then $\max_{i=1,\ldots,n} \varphi(d(s_i, t_i)) = 0$. The purpose of the function φ is to scale the values of the $d(s_i, t_i)$ so that they lie in the interval $[0, 1]$. Thus $\max_{i=1,\ldots,n} \varphi(d(s_i, t_i)) \leq 1$. The factor of $1/2$ means that the greater the 'depth' to which s and t agree, the smaller will be their distance apart. So for lists, for example, the longer the prefix on which two lists agree, the smaller will be their distance apart.

In Part 2 of the definition, for the particular case of sets, $\sum_{r \in supp(s) \cup supp(t)} d(V(s\ r), V(t\ r))$ is the cardinality of the symmetric difference of the sets s and t (assuming that ϱ_Ω is the discrete metric.).

It should be clear that the definition of d does not depend on the choice of α such that $s, t \in \mathfrak{B}_\alpha$. (There may be more than one such α.) What is important is only whether α has the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \cdots \times \alpha_n$, and this is invariant.

The definition given above for d is, of course, only one of a number of possibilities. For example, one could use instead the Euclidean form of the metric (with the square root of the sum of the squares) in Part 3 or a more specialised metric for lists in Part 1. For a particular instance-based learning application, such fine tuning would be almost certainly needed. These variant definitions for d are likely to share the following properties of d; in any case, the proofs of these properties for d show the way for proving similar properties for the variants.

Example 3.6.3. Suppose that ϱ_{Int} is the metric given by $\varrho_{Int}(n, m) = |n - m|$, with a similar definition for ϱ_{Float}. Then $d_{Int}(42, 42) = 0$, $d_{Int}(21, 42) = 21$ and $d_{Float}(42.1, 42.2) = 0.1$.

Example 3.6.4. Suppose that ϱ_{List} is the discrete metric. Let M be a nullary type constructor, $A, B, C, D : M$, and ϱ_M the discrete metric. Suppose that $\varphi(x) = \frac{x}{1+x}$. Let s be the list $[A, B, C]$ and t the list $[A, D]$. (See Fig. 3.2.) Then

$$
\begin{aligned}
d(s,t) &= d([A,B,C],[A,D]) \\
&= \frac{1}{2}\max\{\varphi(d(A,A)), \varphi(d([B,C],[D]))\} \\
&= \frac{1}{2}\varphi(d([B,C],[D])) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(d(B,D)), \varphi(d([C],[]))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(\varrho_M(B,D)), \varphi(\varrho_{List}(\sharp,[]))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(1), \varphi(1)\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\cdot\frac{1}{2}) \\
&= \frac{1}{2}\cdot\frac{\frac{1}{4}}{1+\frac{1}{4}} \\
&= \frac{1}{10}.
\end{aligned}
$$

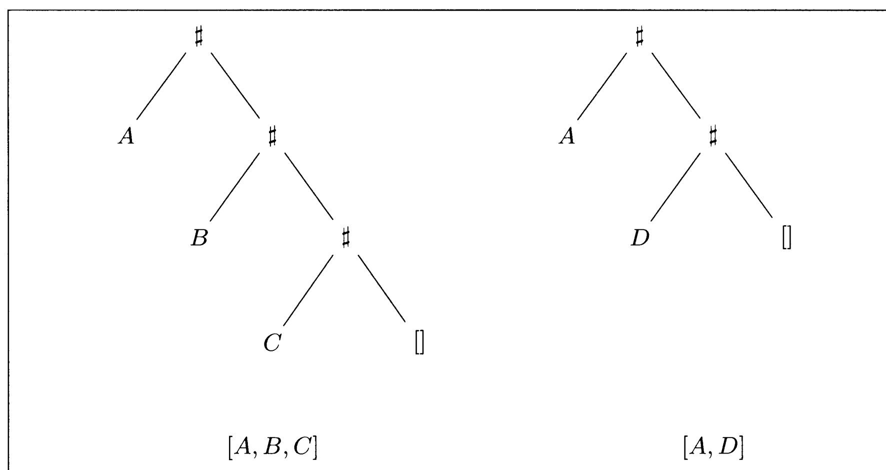

Fig. 3.2. Two lists

Example 3.6.5. Let *BTree* be a unary type constructor, and $Null : BTree\ a$ and $BNode : BTree\ a \to a \to BTree\ a \to BTree\ a$ be data constructors. Here

BTree a is the type of binary trees, *Null* represents the empty binary tree, and *BNode* is used to represent non-empty binary trees. Let ϱ_{BTree} be the discrete metric. Suppose that M is a nullary type constructor, $A, B, C, D : M$, and ϱ_M is the discrete metric. Suppose that $\varphi(x) = \frac{x}{1+x}$. Let s be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ C\ (BNode\ Null\ D\ Null)),$$

a binary tree of type *BTree M*, and t be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ D\ Null).$$

(See Fig. 3.3.) Then

$$\begin{aligned}
& d(s,t) \\
&= \frac{1}{2}\max\{\varphi(d(BNode\ Null\ A\ Null, BNode\ Null\ A\ Null)), \varphi(d(B,B)), \\
&\quad \varphi(d(BNode\ Null\ C\ (BNode\ Null\ D\ Null), BNode\ Null\ D\ Null))\} \\
&= \frac{1}{2}\varphi(d(BNode\ Null\ C\ (BNode\ Null\ D\ Null), BNode\ Null\ D\ Null)) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(d(Null, Null)), \varphi(d(C,D)), \\
&\qquad\qquad \varphi(d(BNode\ Null\ D\ Null, Null))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(\varrho_{BTree}(Null, Null)), \varphi(\varrho_M(C,D)), \\
&\qquad\qquad \varphi(\varrho_{BTree}(BNode, Null))\}) \\
&= \frac{1}{2}\varphi(\frac{1}{2}\max\{\varphi(1), \varphi(1)\}) \\
&= \frac{1}{2}\cdot\varphi(\frac{1}{4}) \\
&= \frac{1}{2}\cdot\frac{\frac{1}{4}}{1+\frac{1}{4}} \\
&= \frac{1}{10}.
\end{aligned}$$

Notation 3.6.1. The basic abstraction $\lambda x.if\ x = t_1\ then\ \top\ else\ \ldots\ if\ x = t_n\ then\ \top\ else\ \bot \in \mathfrak{B}_{\beta\to\Omega}$ is a set whose elements have type more general than β and is denoted by $\{t_1, \ldots, t_n\}$.

Example 3.6.6. Suppose that ϱ_Ω is the discrete metric, M is a nullary type constructor, and $A, B, C, D : M$. If s is the set $\{A, B, C\} \in \mathfrak{B}_{M\to\Omega}$ and t is the set $\{A, D\} \in \mathfrak{B}_{M\to\Omega}$, then $d(s,t) = \sum_{r\in supp(s)\cup supp(t)} d(V(s\ r), V(t\ r)) = 1 + 1 + 1 = 3$.

Notation 3.6.2. The basic abstraction $\lambda x.if\ x = t_1\ then\ m_1\ else\ \ldots\ if\ x = t_n\ then\ m_n\ else\ 0 \in \mathfrak{B}_{\beta\to Nat}$ is a multiset whose elements have type more

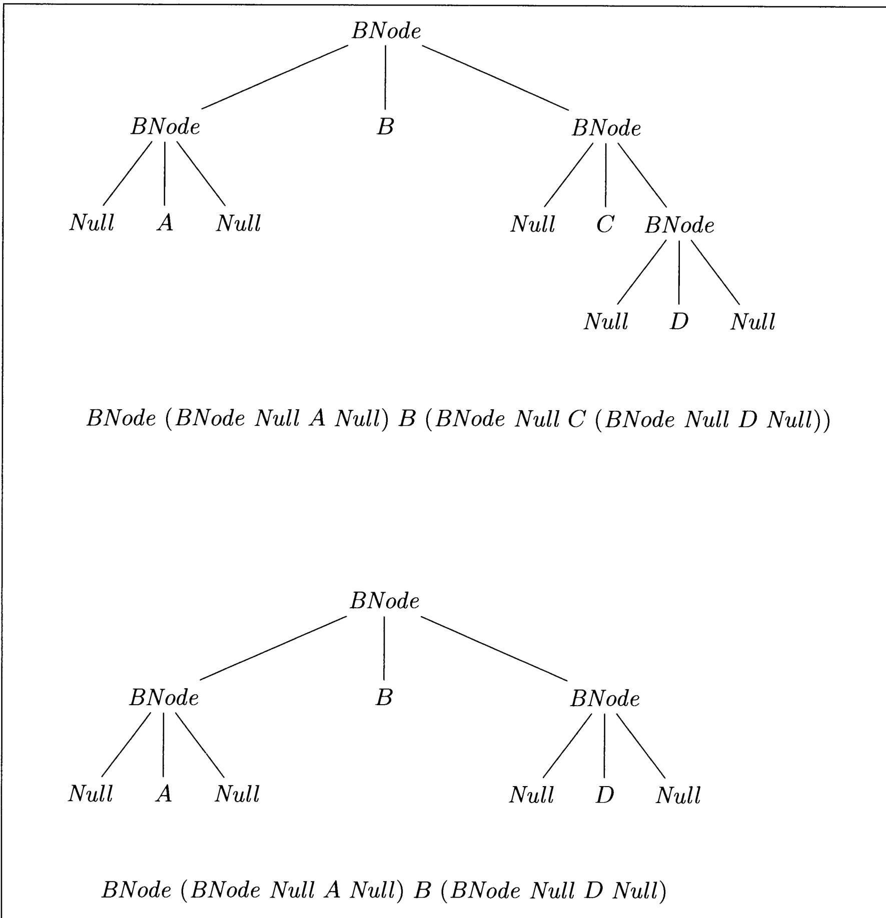

Fig. 3.3. Two binary trees

general than β and is denoted by $\langle t_1, \ldots, t_1, \ldots, t_n, \ldots, t_n \rangle$, where there are m_i occurrences of t_i, for $i = 1, \ldots, n$. (That is, the number of times an element appears in the expression is its multiplicity in the multiset.) Obviously, this notation is only useful for 'small' multisets.

Example 3.6.7. Suppose that ϱ_{Nat} is the metric given by $\varrho_{Nat}(n, m) = |n - m|$, M is a nullary type constructor, and $A, B, C, D : M$. Suppose that s is $\langle A, A, B, C, C, C \rangle \in \mathfrak{B}_{M \to Nat}$ and t is $\langle B, C, C, D \rangle \in \mathfrak{B}_{M \to Nat}$. Then $d(s, t) = \sum_{r \in supp(s) \cup supp(t)} d(V(s\ r), V(t\ r)) = 2 + 1 + 1 = 4$.

Proposition 3.6.1. *Let $d : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ be the function defined in Definition 3.6.3. For each $\alpha \in \mathfrak{S}^c$, $(\mathfrak{B}_\alpha, d)$ is a metric space.*

Proof. It has to be shown that d satisfies the conditions of Definition 3.6.2.

For each $m \in \mathbb{N}$, let $P_1(m)$ be the property:

For each $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $d(s,t) \geq 0$.

It is shown by induction that $P_1(m)$ holds, for all $m \in \mathbb{N}$. The non-negativity of d on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$ (because $\mathfrak{B} = \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$ and $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, for all $m \in \mathbb{N}$, by Proposition 3.5.8).

First it is shown that $P_1(0)$ holds. In this case, s and t are nullary data constructors associated with the same type constructor T, say. By definition, $d(s,t) = \varrho_T(s,t)$ and the result follows because ϱ_T is non-negative.

Now assume that $P_1(m)$ holds. It is proved that $P_1(m+1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $d(s,t) \geq 0$. There are three cases to consider corresponding to α having the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \cdots \times \alpha_m$. In each case, it is easy to see from the definition of d and the induction hypothesis that $d(s,t) \geq 0$. This completes the proof that d is non-negative on each $\mathfrak{B}_\alpha$.

For each $m \in \mathbb{N}$, let $P_2(m)$ be the property:

For each $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $d(s,t) = d(t,s)$.

It is shown by induction that $P_2(m)$ holds, for all $m \in \mathbb{N}$. The symmetry of d on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$.

First it is shown that $P_2(0)$ holds. In this case, s and t are nullary data constructors associated with the same type constructor T, say. By definition, $d(s,t) = \varrho_T(s,t)$ and the result follows because ϱ_T is symmetric.

Now assume that $P_2(m)$ holds. It is proved that $P_2(m+1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $d(s,t) = d(t,s)$. There are three cases to consider corresponding to α having the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \cdots \times \alpha_m$. In each case, it is easy to see from the definition of d and the induction hypothesis that $d(s,t) = d(t,s)$. This completes the proof that d is symmetric on each $\mathfrak{B}_\alpha$.

For each $m \in \mathbb{N}$, let $P_3(m)$ be the property:

For each $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $d(s,t) = 0$ iff $s = t$.

It is shown by induction that $P_3(m)$ holds, for all $m \in \mathbb{N}$. The desired property of d on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$.

First it is shown that $P_3(0)$ holds. In this case, s and t are nullary data constructors associated with the same type constructor T, say. By definition, $d(s,t) = \varrho_T(s,t)$ and the result follows because $\varrho_T(s,t) = 0$ iff $s = t$.

Now assume that $P_3(m)$ holds. It is proved that $P_3(m+1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $d(s,t) = 0$ iff $s = t$. First, I show that $s = t$ implies that $d(s,t) = 0$. In each

of the three cases, it is easy to see from the definition of d and the induction hypothesis that, if $s = t$, then $d(s, t) = 0$.

Conversely, suppose that $d(s, t) = 0$. It has to be shown that $s = t$. As usual, there are three cases to consider.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then s is $C\ s_1 \ldots s_n$ and t is $C\ t_1 \ldots t_n$, for some C. If $n = 0$, then $s = t$. If $n > 0$, then $\varphi(d(s_i, t_i)) = 0$, for $i = 1, \ldots, n$. Thus, by the properties of φ, $d(s_i, t_i) = 0$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.
2. Let $\alpha = \beta \to \gamma$. Then $d(V(s\ r), V(t\ r)) = 0$, for all $r \in \mathit{supp}(s) \cup \mathit{supp}(t)$. By the induction hypothesis, $V(s\ r) = V(t\ r)$, for all $r \in \mathit{supp}(s) \cup \mathit{supp}(t)$. Hence $V(s\ r) = V(t\ r)$, for all $r \in \mathfrak{B}_\beta$. Thus $s = t$, by Propositions 3.3.3 and 3.5.9.
3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and $d(s_i, t_i) = 0$, for $i = 1, \ldots, n$. By the induction hypothesis, $s_i = t_i$, for $i = 1, \ldots, n$. Thus $s = t$.

This completes the proof that $d(s, t) = 0$ iff $s = t$ on each $\mathfrak{B}_\alpha$.

It only remains to prove the triangle inequality. For each $m \in \mathbb{N}$, let $P_4(m)$ be the property:

> For each $\alpha \in \mathfrak{S}^c$ and $r, s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $d(r, t) \leq d(r, s) + d(s, t)$.

It is shown by induction that $P_4(m)$ holds, for all $m \in \mathbb{N}$. The triangle inequality for d on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $r, s, t \in \mathfrak{B}_\alpha$, there exists an m such that $r, s, t \in \mathfrak{B}_m$.

First it is shown that $P_4(0)$ holds. In this case, r, s and t are nullary data constructors associated with the same type constructor T, say. By definition, $d(r, t) = \varrho_T(r, t)$, and so on, and the result follows because ϱ_T satisfies the triangle inequality.

Now assume that $P_4(m)$ holds. It is proved that $P_4(m + 1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $r, s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $d(r, t) \leq d(r, s) + d(s, t)$. There are three cases to consider.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Then r is $B\ r_1 \ldots r_p$, s is $C\ s_1 \ldots s_n$, and t is $D\ t_1 \ldots t_m$. If there is no data constructor associated with T of arity > 0, then $p = n = m = 0$ and the inequality holds because ϱ_T is a metric. Otherwise, there is a data constructor of arity > 0 and ϱ_T is the discrete metric. Suppose first that B, C, and D are not all the same. Thus at least one of $d(r, s)$ or $d(s, t)$ is 1, and the inequality holds since $d(r, t) \leq 1$. Now suppose that $B = C = D$. Then $p = n = m$. By the induction hypothesis, $d(r_i, t_i) \leq d(r_i, s_i) + d(s_i, t_i)$, for $i = 1, \ldots, n$. Hence $\max_{i=1,\ldots,n} \varphi(d(r_i, t_i)) \leq \max_{i=1,\ldots,n} \varphi(d(r_i, s_i)) + \max_{i=1,\ldots,n} \varphi(d(s_i, t_i))$, by the properties of φ, and so $d(r, t) \leq d(r, s) + d(s, t)$.

2. Let $\alpha = \beta \to \gamma$. By the induction hypothesis, $d(V(r\ b), V(t\ b)) \leq d(V(r\ b), V(s\ b)) + d(V(s\ b), V(t\ b))$, for all $b \in \mathfrak{B}_\beta$. Also $b \notin supp(s) \cup supp(t)$ implies $d(V(s\ b), V(t\ b)) = d(s_0, s_0) = 0$, where $s_0 \in \mathfrak{D}\gamma$. Put $R = supp(r)$, $S = supp(t)$ and $T = supp(t)$. Then

$$\begin{aligned}
&d(r,t) \\
&= \sum_{b \in R \cup T} d(V(r\ b),\ V(t\ b)) \\
&= \sum_{b \in R \cup S \cup T} d(V(r\ b),\ V(t\ b)) \\
&\leq \sum_{b \in R \cup S \cup T} d(V(r\ b),\ V(s\ b)) + \sum_{b \in R \cup S \cup T} d(V(s\ b),\ V(t\ b)) \\
&= \sum_{b \in R \cup S} d(V(r\ b),\ V(s\ b)) + \sum_{b \in S \cup T} d(V(s\ b),\ V(t\ b)) \\
&= d(r,s) + d(s,t).
\end{aligned}$$

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Then r is $(r_1, \ldots, r_n)$, s is $(s_1, \ldots, s_n)$, and t is $(t_1, \ldots, t_n)$. By the induction hypothesis, $d(r_i, t_i) \leq d(r_i, s_i) + d(s_i, t_i)$, for $i = 1, \ldots, n$. Hence $d(r,t) = \sum_{i=1}^n d(r_i, t_i) \leq \sum_{i=1}^n d(r_i, s_i) + \sum_{i=1}^n d(s_i, t_i) = d(r,s) + d(s,t)$. □

The function d is not generally a metric on the *whole* of $\mathfrak{B}$.

Example 3.6.8. Suppose that ϱ_{Float} is the metric given by $\varrho_{Float}(n, m) = |n - m|$ and that 0.0 is the default data constructor for *Float*. Let T_1 and T_2 be nullary type constructors, $A_1 : T_1$ and $A_2 : T_2$ be data constructors, r be $\lambda x.if\ x = A_1\ then\ 0.1\ else\ 0.0$, s be $\lambda x.\ 0.0$, and t be $\lambda x.if\ x = A_2\ then\ 0.1\ else\ 0.0$. Then $r, s \in \mathfrak{B}_{T_1 \to Float}$ and $d(r,s) = 0.1$. Furthermore, $s, t \in \mathfrak{B}_{T_2 \to Float}$ and $d(s,t) = 0.1$. But $d(r,t) = 1$, since there does not exist $\alpha \in \mathfrak{S}^c$ such that $r, t \in \mathfrak{B}_\alpha$. Thus d on $\mathfrak{B}$ does not satisfy the triangle inequality, since $d(r,t) \not\leq d(r,s) + d(s,t)$.

Of course, the definition that $d(r,t) = 1$ for r and t having incompatible types is arbitrary. But, whatever the choice of $d(r,t)$ for such r and t, it must be > 0 and the argument of the example shows that the triangle equality still does not hold. (It may be necessary to replace the value 0.1 in r and t by a smaller quantity.)

The metric d of Definition 3.6.3 may be unbounded. A standard technique provides a bounded metric, in fact, one whose values lie in $[0,1]$.

Proposition 3.6.2. *Let φ be a non-decreasing function from the non-negative reals into the closed interval $[0,1]$ such that $\varphi(0) = 0$, $\varphi(x) > 0$ if $x > 0$, and $\varphi(x+y) \leq \varphi(x) + \varphi(y)$, for each x and y. Let $(\mathfrak{X}, d)$ be a metric space. Then the function d' defined by $d'(x,y) = \varphi(d(x,y))$, for all $x, y \in \mathfrak{X}$, is a bounded metric on $\mathfrak{X}$.*

Proof. Straightforward. □

Part 2 of Definition 3.6.3 can be written more neatly.

Proposition 3.6.3. *If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then $d(s,t) = \sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r))$.*

Proof. Let $s_0 \in \mathfrak{D}_\gamma$. Then $d(V(s\ r), V(t\ r)) = d(s_0, s_0) = 0$, for all $r \notin supp(s) \cup supp(t)$, by Proposition 3.6.1. It follows that $d(s,t) = \sum_{r \in supp(s) \cup supp(t)} d(V(s\ r), V(t\ r)) = \sum_{r \in \mathfrak{B}_\beta} d(V(s\ r), V(t\ r))$. □

The generalised definition of cardinality for basic abstractions is now given.

Definition 3.6.4. Let t be $\lambda x.$*if* $x = t_1$ *then* s_1 *else* ... *if* $x = t_n$ *then* s_n *else* $s_0 \in \mathfrak{B}_{\beta \to \gamma}$. Then the *cardinality* of t, $card(t)$, is defined by

$$card(t) = \sum_{r \in supp(t)} d(V(t\ r),\ s_0).$$

The function *card* measures how much a basic abstraction deviates from being constant.

Example 3.6.9. Suppose that ϱ_Ω is the discrete metric. If t is the set $\{A, B, C\}$, then $card(t) = 3$. That is, $card(t)$ is the cardinality of the set t.

Example 3.6.10. Suppose that ϱ_{Nat} is the metric given by $\varrho_{Nat}(n,m) = |n - m|$. If t is the multiset $\langle A, B, A, B, C \rangle$, then $card(t) = 2 + 2 + 1 = 5$. That is, $card(t)$ is the sum of the multiplicities of the elements of the multiset t.

Proposition 3.6.4. *Let t be $\lambda x.$if $x = t_1$ then s_1 else ... if $x = t_n$ then s_n else $s_0 \in \mathfrak{B}_{\beta \to \gamma}$. Then $card(t) = \sum_{r \in \mathfrak{B}_\beta} d(V(t\ r),\ s_0) = d(t, \lambda x.s_0)$.*

Proof. For the first equality, note that $d(V(t\ r),\ s_0) = d(s_0, s_0) = 0$, for all $r \notin supp(t)$. For the second, note that $V(\lambda x.s_0\ r) = s_0$, for all $r \in \mathfrak{B}_\beta$, and $supp(\lambda x.s_0) = \{\}$. □

Typically, metric-based learning methods are generic in the sense that they only require that it be possible to define the distance between pairs of individuals and are independent of the nature of the individuals. Thus the results of this section can be applied directly to a variety of metric-based learning methods for structured data by simply using the metrics developed here in these algorithms.

3.7 Kernels on Basic Terms

Learning methods that rely on kernels are becoming increasingly widely used. This section provides kernels for basic terms that opens up the way for kernel-based learning methods to be applied to individuals that can be represented by basic terms.

The starting point is the definition of a kernel.

Definition 3.7.1. Let $\mathcal{X}$ be a set. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is *positive definite* on $\mathcal{X}$ if, for all $n \in \mathbb{Z}^+, x_1, \ldots, x_n \in \mathcal{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) \geq 0$.

Note that the x_i are not necessarily distinct. If the condition that the x_i must be distinct is added to the definition above, it is easy to see that the two definitions are equivalent. Note also that if k is positive definite, then $k(x, x) \geq 0$, for all $x \in \mathcal{X}$.

Definition 3.7.2. Let $\mathcal{X}$ be a set. A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *kernel* if it is symmetric and positive definite.

Of course, symmetric means $k(x, y) = k(y, x)$, for all $x, y \in \mathcal{X}$.

One can think of a kernel as being a generalised inner product, and therefore a 'similarity' measure, between individuals. Many learning algorithms, for example, support vector machines, depend only on being able to compute the inner product (also called dot product or scalar product) between individuals. Originally, individuals were actually represented by vectors in $\mathbb{R}^m$ and the usual inner product in $\mathbb{R}^m$ was used in these algorithms. However, recent versions of these algorithms substitute a kernel for the inner product.

The justification for this replacement is as follows. Let $\mathcal{X}$ be a set and k a kernel on $\mathcal{X}$. Then there exists a Hilbert space $\mathcal{H}$ and a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ such that $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$, where $\langle \cdot, \cdot \rangle$ is the inner product in $\mathcal{H}$. (See Proposition 3.7.2 below.) The mapping Φ may be non-linear – in fact, unless $\mathcal{X}$ is a linear space, one cannot even ask if Φ is linear. This means that any set, whether a linear space or not, that admits a kernel can be embedded into a linear space. Consequently, one can then 'add' elements of the set or 'multiply' them by a scalar. (Of course, the addition and scalar multiplication is taking place on the images of the elements under Φ.)

From a learning point of view, the space $\mathcal{H}$ is a *feature space* for the individuals and Φ maps each individual into its vector of features. One of the attractive aspects of the kernel approach is that it is not necessary to calculate Φ or $\mathcal{H}$ – they are entirely implicit. The learning algorithms only ever need to be able to calculate $k(x, x')$, for any x and x' in $\mathcal{X}$.

In preparation for the definition of the kernel on $\mathfrak{B}$, here are kernels on some basic data types.

Example 3.7.1. For numerical types, such as *Nat*, *Int*, and *Float*, the product function is a kernel, called the *product kernel.*

For sets of data constructors without any other structure, the following kernel is generally used.

Definition 3.7.3. Let $\mathfrak{X}$ be a set. The *discrete kernel* $\delta : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ is defined by

$$\delta(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise.} \end{cases}$$

Clearly δ is a kernel, as

$$\sum_{i,j \in \{1,\ldots,n\}} c_i \, c_j \, \delta(x_i, x_j) = \sum_{i \in \{1,\ldots,n\}} c_i^2 \geq 0.$$

Example 3.7.2. For the data constructors $\sharp$ and $[]$ associated with the type constructor *List*, the discrete kernel δ is given by $\delta(\sharp, \sharp) = 1$, $\delta(\sharp, []) = 0$, $\delta([], \sharp) = 0$, and $\delta([], []) = 1$.

Example 3.7.3. Let $\mathfrak{X}$ be a set and $f : \mathfrak{X} \to \mathbb{R}$ any real-valued function. Then $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ defined by $k(x, y) = f(x)f(y)$ is a kernel.

Example 3.7.4. Let $k : \mathbb{R}^n \to \mathbb{R}$ be defined by $k(x, y) = \exp(-\|x - y\|^2/\sigma^2)$, where $\|\cdot\|$ is the Euclidean norm on $\mathbb{R}^n$ and $\sigma > 0$. Then k is a kernel, called the Gaussian kernel.

Next the kernel on $\mathfrak{B}$ is defined. The following definition of a kernel on basic terms assumes the existence of kernels on the various sets of data constructors. More precisely, it is assumed that there are functions κ_T, for $T \in \mathfrak{T}$, satisfying the following conditions.

1. For each type constructor $T \in \mathfrak{T}$, κ_T is assumed to be a kernel on the set of data constructors associated with T.
2. If there is at least one data constructor of arity > 0 associated with T, then κ_T is the discrete kernel.

Definition 3.7.4. The function $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively as follows. Let $s, t \in \mathfrak{B}$.

1. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, then

$$k(s, t) = \begin{cases} \kappa_T(C, D) & \text{if } C \neq D \\ \kappa_T(C, C) + \sum_{i=1}^{n} k(s_i, t_i) & \text{otherwise,} \end{cases}$$

where s is $C\ s_1 \ldots s_n$ and t is $D\ t_1 \ldots t_m$.

2. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then

$$k(s,t) = \sum_{u \in supp(s) \cap supp(t)} k(V(s\ u), V(t\ u)).$$

3. If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, then

$$k(s,t) = \sum_{i=1}^{n} k(s_i, t_i),$$

   where s is $(s_1, \ldots, s_n)$ and t is $(t_1, \ldots, t_n)$.
4. If there does not exist $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{B}_\alpha$, then $k(s,t) = 0$.

The well-definedness of the function k depends upon Proposition A.1.4: k is defined directly on the minimal elements (that is, pairs of the form (C, D), where C and D are nullary data constructors) and, for other pairs, is uniquely determined by the rules for each of the three kinds of basic terms and Part 4 when the types do not match.

The definition for k in Definition 3.7.4 is, of course, only one of many possibilities, but it at least establishes the general form of kernels on $\mathfrak{B}$. See the exercises at the end of the chapter for some of the other possibilities. Furthermore, the proof that k is a kernel in Proposition 3.7.1 below provides the general approach to proving these alternative functions are also kernels.

It should be clear that the definition of k does not depend on the choice of α such that $s, t \in \mathfrak{B}_\alpha$. (There may be more than one such α.) What is important is only whether α has the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \to \gamma$, or $\alpha_1 \times \cdots \times \alpha_n$, and this is invariant.

Note 3.7.1. There are several special cases of the preceding definition that are of interest. First, if α is $\beta \to \Omega$ and $\bot$ is the default data constructor for Ω, then the abstractions of Part 2 of the definition are, of course, finite sets. Let κ_Ω be the discrete kernel. Then

$$k(s,t) = |supp(s) \cap supp(t)|,$$

since $k(V(s\ u), (V(t\ u)) = \kappa_\Omega(\top, \top) = 1$, for $u \in supp(s) \cap supp(t)$. ($|A|$ is the cardinality of the set A.) If s or t is $\lambda x.\bot$, then $k(s,t) = 0$. More generally, if $supp(s) \cap supp(t) = \emptyset$, then $k(s,t) = 0$.

Let α be $Float \times \cdots \times Float$, where there are n components in the product. Denote this type by $Float^n$. Let κ_{Float} be the product kernel. Then the kernel in Part 3 for $\mathfrak{B}_{Float^n}$ is simply the usual inner product on $\mathbb{F}^n$.

Several examples illustrating how to compute the kernel on individuals of certain types are now given.

Example 3.7.5. Suppose that κ_{List} is the discrete kernel. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_M is the discrete kernel. Let s be the list $[A, B, C]$ and t the list $[A, D]$. (See Fig. 3.2.) Then

$$\begin{aligned} k(s,t) &= \kappa_{List}(\sharp,\sharp) + k(A,A) + k([B,C],[D]) \\ &= 1 + \kappa_M(A,A) + \kappa_{List}(\sharp,\sharp) + k(B,D) + k([C],[]) \\ &= 1+1+1+\kappa_M(B,D) + \kappa_{List}(\sharp,[]) \\ &= 3+0+0 \\ &= 3. \end{aligned}$$

Example 3.7.6. Let *BTree* be a unary type constructor, and $Null : BTree\ a$ and $BNode : BTree\ a \to a \to BTree\ a \to BTree\ a$ be data constructors. Suppose that κ_{BTree} is the discrete kernel. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_M is the discrete kernel. Let s be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ C\ (BNode\ Null\ D\ Null)),$$

a binary tree of type $BTree\ M$, and t be

$$BNode\ (BNode\ Null\ A\ Null)\ B\ (BNode\ Null\ D\ Null).$$

(See Fig. 3.3.) Then

$$\begin{aligned} k(s,t) &= \kappa_{BTree}(BNode, BNode) \\ &\quad + k(BNode\ Null\ A\ Null, BNode\ Null\ A\ Null) + k(B,B) \\ &\quad + k(BNode\ Null\ C\ (BNode\ Null\ D\ Null), BNode\ Null\ D\ Null) \\ &= 1 + \kappa_{BTree}(BNode, BNode) + k(Null, Null) + k(A,A) \\ &\quad + k(Null, Null) + \kappa_M(B,B) + \kappa_{BTree}(BNode, BNode) \\ &\quad + k(Null, Null) + k(C,D) + k(BNode\ Null\ D\ Null, Null) \\ &= 1+1+1+1+1+1+1+1+0+\kappa_{BTree}(BNode, Null) \\ &= 8. \end{aligned}$$

Example 3.7.7. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_Ω is the discrete kernel. If s is the set $\{A, B, C\} \in \mathfrak{B}_{M \to \Omega}$ and t is the set $\{A, C, D\} \in \mathfrak{B}_{M \to \Omega}$, then

$$\begin{aligned} k(s,t) &= k(V(s\ A), V(t\ A)) + k(V(s\ C), V(t\ C)) \\ &= \kappa_\Omega(\top,\top) + \kappa_\Omega(\top,\top) \\ &= 1+1 \\ &= 2. \end{aligned}$$

Example 3.7.8. Let M be a nullary type constructor and $A, B, C, D : M$. Suppose that κ_{Nat} is the product kernel. If s is $\langle A, A, B, C, C, C \rangle \in \mathfrak{B}_{M \to Nat}$ and t is $\langle B, B, C, C, D \rangle \in \mathfrak{B}_{M \to Nat}$, then

$$\begin{aligned} k(s,t) &= k(V(s\ B), V(t\ B)) + k(V(s\ C), V(t\ C)) \\ &= \kappa_{Nat}(1,2) + \kappa_{Nat}(3,2) \\ &= 1 \times 2 + 3 \times 2 \\ &= 8. \end{aligned}$$

Proposition 3.7.1. *Let $k : \mathfrak{B} \times \mathfrak{B} \rightarrow \mathbb{R}$ be the function defined in Definition 3.7.4. Then k is a kernel on $\mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.*

Proof. First the symmetry of k on each $\mathfrak{B}_\alpha$ is established. For each $m \in \mathbb{N}$, let $P_1(m)$ be the property:

For each $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, it follows that $k(s,t) = k(t,s)$.

It is shown by induction that $P_1(m)$ holds, for all $m \in \mathbb{N}$. The symmetry of k on each $\mathfrak{B}_\alpha$ follows immediately from this since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$ (because $\mathfrak{B} = \bigcup_{m \in \mathbb{N}} \mathfrak{B}_m$ and $\mathfrak{B}_m \subseteq \mathfrak{B}_{m+1}$, for all $m \in \mathbb{N}$, by Proposition 3.5.8).

First it is shown that $P_1(0)$ holds. In this case, s and t are nullary data constructors associated with the same type constructor T, say. By definition, $k(s,t) = \kappa_T(s,t)$ and the result follows because κ_T is symmetric.

Now assume that $P_1(m)$ holds. It is proved that $P_1(m+1)$ also holds. Thus suppose that $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. It has to be shown that $k(s,t) = k(t,s)$. There are three cases to consider corresponding to α having the form $T\ \alpha_1 \ldots \alpha_k$, $\beta \rightarrow \gamma$, or $\alpha_1 \times \cdots \times \alpha_m$. In each case, it is easy to see from the definition of k and the induction hypothesis that $k(s,t) = k(t,s)$. This completes the proof that k is symmetric on each $\mathfrak{B}_\alpha$.

For the remaining part of the proof, for each $m \in \mathbb{N}$, let $P_2(m)$ be the property:

For each $n \in \mathbb{Z}^+, \alpha \in \mathfrak{S}^c, t_1, \ldots, t_n \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j \in \{1,\ldots,n\}} c_i\ c_j\ k(t_i, t_j) \geq 0$.

It is shown by induction that $P_2(m)$ holds, for all $m \in \mathbb{N}$. The remaining condition for positive definiteness follows immediately from this since, given $t_1, \ldots, t_n \in \mathfrak{B}_\alpha$, there exists an m such that $t_1, \ldots, t_n \in \mathfrak{B}_m$.

First it is shown that $P_2(0)$ holds. In this case, each t_i is a nullary data constructor associated with the same type constructor T, say. By definition, $k(t_i, t_j) = \kappa_T(t_i, t_j)$, for each i and j, and the result follows since κ_T is assumed to be positive definite.

Now assume that $P_2(m)$ holds. It is proved that $P_2(m+1)$ also holds. Thus suppose that $n \in \mathbb{Z}^+, \alpha \in \mathfrak{S}^c, t_1, \ldots, t_n \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, and $c_1, \ldots, c_n \in \mathbb{R}$. It has to be shown that $\sum_{i,j \in \{1,\ldots,n\}} c_i\ c_j\ k(t_i, t_j) \geq 0$. There are three cases to consider.

1. Let $\alpha = T\ \alpha_1 \ldots \alpha_k$. Suppose that $t_i = C_i\ t_i^{(1)} \ldots t_i^{(m_i)}$, where $m_i \geq 0$, for $i = 1, \ldots, n$. Let $\mathcal{C} = \{C_i \mid i = 1, \ldots, n\}$. Then

$$
\begin{aligned}
&\sum_{i,j\in\{1,\dots,n\}} c_i\, c_j\, k(t_i,t_j)\\
&= \sum_{i,j\in\{1,\dots,n\}} c_i\, c_j\, \kappa_T(C_i,C_j)\\
&\quad + \sum_{\substack{i,j\in\{1,\dots,n\}\\ C_i=C_j}} c_i\, c_j \sum_{l\in\{1,\dots,arity(C_i)\}} k(t_i^{(l)},t_j^{(l)}).
\end{aligned}
$$

Now $\sum_{i,j\in\{1,\dots,n\}} c_i\, c_j\, \kappa_T(C_i,C_j) \geq 0$, using the fact that κ_T is a kernel on the set of data constructors associated with T. Also

$$
\begin{aligned}
&\sum_{\substack{i,j\in\{1,\dots,n\}\\ C_i=C_j}} c_i\, c_j \sum_{l\in\{1,\dots,arity(C_i)\}} k(t_i^{(l)},t_j^{(l)})\\
&= \sum_{C\in\mathfrak{C}} \sum_{\substack{i,j\in\{1,\dots,n\}\\ C_i=C_j=C}} \sum_{l\in\{1,\dots,arity(C)\}} c_i\, c_j\, k(t_i^{(l)},t_j^{(l)})\\
&= \sum_{C\in\mathfrak{C}} \sum_{l\in\{1,\dots,arity(C)\}} \sum_{\substack{i,j\in\{1,\dots,n\}\\ C_i=C_j=C}} c_i\, c_j\, k(t_i^{(l)},t_j^{(l)})\\
&\geq 0,
\end{aligned}
$$

by the induction hypothesis.

2. Let $\alpha = \beta \to \gamma$. Then

$$
\begin{aligned}
&\sum_{i,j\in\{1,\dots,n\}} c_i\, c_j\, k(t_i,t_j)\\
&= \sum_{i,j\in\{1,\dots,n\}} c_i\, c_j \sum_{u\in supp(t_i)\cap supp(t_j)} k(V(t_i\; u),V(t_j\; u))\\
&= \sum_{i,j\in\{1,\dots,n\}} \sum_{u\in supp(t_i)\cap supp(t_j)} c_i\, c_j\, k(V(t_i\; u),V(t_j\; u))\\
&= \sum_{i,j\in\{1,\dots,n\}} \sum_{u\in\mathfrak{B}_\beta} c'_{i,u}\, c'_{j,u}\, k(V(t_i\; u),V(t_j\; u))\\
&= \sum_{u\in\mathfrak{B}_\beta} \sum_{i,j\in\{1,\dots,n\}} c'_{i,u}\, c'_{j,u}\, k(V(t_i\; u),V(t_j\; u))\\
&\geq 0,
\end{aligned}
$$

by the induction hypothesis, where

$$
c'_{i,u} = \begin{cases} c_i & u \in supp(t_i)\\ 0 & u \notin supp(t_i). \end{cases}
$$

3. Let $\alpha = \alpha_1 \times \cdots \times \alpha_m$. Suppose that $t_i = (t_i^{(1)}, \ldots, t_i^{(m)})$, for $i = 1, \ldots, n$. Then

$$\begin{aligned}
&\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(t_i, t_j) \\
&= \sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j (\sum_{l=1}^{m} k(t_i^{(l)}, t_j^{(l)})) \\
&= \sum_{l=1}^{m} \sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(t_i^{(l)}, t_j^{(l)}) \\
&\geq 0,
\end{aligned}$$

by the induction hypothesis. □

Attention is now focussed on the general properties of kernels. Earlier I mentioned that sets that admit a kernel can be embedded in a Hilbert space. Here is the relevant result.

Proposition 3.7.2. *Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel on $\mathcal{X}$. Then there exists a Hilbert space $\mathcal{H}$ and a function $\Phi : \mathcal{X} \to \mathcal{H}$ such that $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$, where $\langle \cdot, \cdot \rangle$ is the inner product in $\mathcal{H}$.*

Proof. In outline, the proof proceeds as follows. Define $\Phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}$ by $\Phi(x) = k(\cdot, x)$, for each $x \in \mathcal{X}$. Let $\mathcal{K} \subset \mathbb{R}^{\mathcal{X}}$ be the space of all linear combinations of functions of the form $k(\cdot, x)$, for some x. An inner product $\langle \cdot, \cdot \rangle$ on $\mathcal{K}$ is defined as follows: if $f = \sum_{i=1}^{n} \alpha_i k(\cdot, x_i)$ and $g = \sum_{j=1}^{m} \beta_j k(\cdot, x'_j)$, then

$$\langle f, g \rangle = \sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_i \beta_j k(x_i, x'_j).$$

Note that, since k is positive definite, $\langle f, f \rangle = \sum_{i,j=1}^{n} \alpha_i \alpha_j k(x_i, x_j) \geq 0$. One can show that $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$. The Hilbert space $\mathcal{H}$ is the completion of $\mathcal{K}$. □

The function Φ in Proposition 3.7.2 is called the *associated embedding* for k. Also the Hilbert space $\mathcal{H}$ is called the *associated Hilbert space* for k.

Conversely, if Φ is a mapping from $\mathcal{X}$ into some Hilbert space $\mathcal{H}$ (over the reals), then it is easy to define a kernel on $\mathcal{X}$ using Φ.

Proposition 3.7.3. *Let $\Phi : \mathcal{X} \to \mathcal{H}$ be a function from a set $\mathcal{X}$ into a Hilbert space $\mathcal{H}$ (over the reals). Then $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ defined by $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$, is a kernel on $\mathcal{X}$.*

Proof. Let $x, y \in \mathcal{X}$. Then $k(x,y) = \langle \Phi(x), \Phi(y) \rangle = \langle \Phi(y), \Phi(x) \rangle = k(y,x)$, by the symmetry of $\langle \cdot, \cdot \rangle$. Thus k is symmetric.

Let $n \in \mathbb{Z}^+, x_1, \ldots, x_n \in \mathcal{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$. Then

$$
\begin{aligned}
& \sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) \\
&= \left\langle \sum_{i \in \{1,\ldots,n\}} c_i \Phi(x_i), \sum_{j \in \{1,\ldots,n\}} c_j \Phi(x_j) \right\rangle \\
&= \left\| \sum_{i \in \{1,\ldots,n\}} c_i \Phi(x_i) \right\|^2 \\
&\geq 0.
\end{aligned}
$$

Thus k is positive definite. □

It follows from Propositions 3.7.2 and 3.7.3 that a function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a kernel iff there exists a function $\Phi : \mathcal{X} \to \mathcal{H}$, where $\mathcal{H}$ is a Hilbert space, such that $k(x,y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$.

It is of considerable interest to know when Φ is injective. For kernel-based learning methods this is important since, if Φ were not injective, there would be no way of assigning different classes, for example, to two distinct individuals s and t for which $\Phi(s) = \Phi(t)$.

Definition 3.7.5. A kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is *separating* if $x \neq y$ implies $k(\cdot, x) \neq k(\cdot, y)$, for all $x, y \in \mathcal{X}$.

In other words, k is separating if $x \neq y$ implies there exists $z \in \mathcal{X}$ such that $k(z,x) \neq k(z,y)$, for all $x, y \in \mathcal{X}$.

Example 3.7.9. The discrete kernel δ is separating: if $x \neq y$, then $\delta(z,x) \neq \delta(z,y)$, for $z = x$, say.

The product kernel is also separating: just take $z = 1$.

Proposition 3.7.4. *Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel and Φ the associated embedding. Then the following are equivalent.*

1. *k is separating.*
2. *Φ is injective.*
3. *$x \neq y$ implies $k(x,x) - 2k(x,y) + k(y,y) \neq 0$, for all $x, y \in \mathcal{X}$.*

Proof. By the proof of Proposition 3.7.2, Φ is defined by $\Phi(x) = k(\cdot, x)$, for $x \in \mathcal{X}$. Thus it is clear that k is separating iff Φ is injective.

By Proposition 3.7.2, $k(x,y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$. Hence

$$
\begin{aligned}
& k(x,x) - 2k(x,y) + k(y,y) \\
&= \langle \Phi(x), \Phi(x) \rangle - 2\langle \Phi(x), \Phi(y) \rangle + \langle \Phi(y), \Phi(y) \rangle \\
&= \langle \Phi(x) - \Phi(y), \Phi(x) - \Phi(y) \rangle.
\end{aligned}
$$

Thus Φ is injective iff $x \neq y$ implies $k(x,x) - 2k(x,y) + k(y,y) \neq 0$, for all $x, y \in \mathfrak{X}$. $\square$

The next task is to show that, under some natural conditions, the kernel k on the $\mathfrak{B}_\alpha$ given in Definition 3.7.4 is separating.

Definition 3.7.6. The kernel κ_T on the set of data constructors associated with a type constructor T is *default-consistent* if $\kappa_T(C,C) > 0$, for each non-default data constructor C associated with T.

Example 3.7.10. Suppose that 0 is the default data constructor for the floating-point numbers. Then the product kernel and the Gaussian kernel on the floating-point numbers are both default-consistent. The discrete kernel on the set of data constructors associated with some type constructor is default-consistent, whatever the choice of default data constructor.

Proposition 3.7.5. *Suppose, for each type constructor T, the kernel κ_T on the set of data constructors associated with T is default-consistent. Let k be the kernel of Definition 3.7.4. Then, for each $\alpha \in \mathfrak{S}^c$ and $s \in \mathfrak{B}_\alpha$, $s \notin \mathfrak{D}$ implies $k(s,s) > 0$.*

Proof. For each $m \in \mathbb{N}$, let $P(m)$ be the property:

For each $\alpha \in \mathfrak{S}^c$ and $s \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$, $s \notin \mathfrak{D}$ implies $k(s,s) > 0$.

It is shown by induction that $P(m)$ holds, for all $m \in \mathbb{N}$. The fact that k has the required property follows immediately since, given $s \in \mathfrak{B}_\alpha$, there exists an m such that $s \in \mathfrak{B}_m$.

First I show that $P(0)$ holds. Let $\alpha \in \mathfrak{S}^c$ and $s \in \mathfrak{B}_\alpha \cap \mathfrak{B}_0$, where $s \notin \mathfrak{D}$. Thus s is a non-default, nullary data constructor associated with some type constructor T, where α has the form $T\ \alpha_1 \ldots \alpha_k$, for some $\alpha_1, \ldots, \alpha_k$ and $k \in \mathbb{N}$. Since κ_T is default-consistent, $k(s,s) = \kappa_T(s,s) > 0$. Thus $P(0)$ is proved.

Now assume that $P(m)$ holds. I prove that $P(m+1)$ also holds. There are three cases to consider.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$. Suppose that $s \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where $s \notin \mathfrak{D}$. Let s be $C\ s_1 \ldots s_n$, where C has the signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $\xi = \{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$. Then $k(s,s) = \kappa_T(C,C) + \sum_{i=1}^{n} k(s_i, s_i)$. Since $s \notin \mathfrak{D}$, either C is a non-default data constructor or some $s_j \notin \mathfrak{D}$. Suppose first that C is not a default data constructor. Then $\kappa_T(C,C) > 0$, since κ_T is default-consistent, and each $k(s_i, s_i) \geq 0$. Thus $k(s,s) > 0$. Suppose next that $s_j \notin \mathfrak{D}$. Now $s_j \in \mathfrak{B}_{\sigma_j \xi} \cap \mathfrak{B}_m$, by Proposition 3.5.5. Thus $k(s_j, s_j) > 0$, by the induction hypothesis. Hence $k(s,s) > 0$.
2. $\alpha = \beta \to \gamma$. Suppose that $s \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where $s \notin \mathfrak{D}$. Then there exists $u \in \mathfrak{B}_\beta$ such that $V(s\ u) \in \mathfrak{B}_\gamma \cap \mathfrak{B}_m$, by Proposition 3.5.6, and $V(s\ u) \notin \mathfrak{D}$, since $s \notin \mathfrak{D}$. By the induction hypothesis, $k(V(s\ u), V(s\ u)) > 0$. It follows that $k(s,s) > 0$.

3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Suppose that $s \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where s is $(s_1, \ldots, s_n)$ and $s \notin \mathfrak{D}$. Since $s \notin \mathfrak{D}$, there exists some j such that $s_j \notin \mathfrak{D}$ and, by Proposition 3.5.5, $s_j \in \mathfrak{B}_{\alpha_j} \cap \mathfrak{B}_m$. Thus, by the induction hypothesis, $k(s_j, s_j) > 0$. It follows that $k(s, s) > 0$. □

Definition 3.7.7. The kernel κ_T on the set of data constructors associated with a type constructor T is *constructor-separating* if, for all nullary data constructors C and D associated with T such that $C \neq D$, there exists a nullary data constructor E associated with T such that $\kappa_T(E, C) \neq \kappa_T(E, D)$.

Example 3.7.11. The product kernel and the Gaussian kernel on the floating-point numbers are both constructor-separating. The discrete kernel on the set of data constructors associated with some type constructor is constructor-separating.

Proposition 3.7.6. *Suppose, for each type constructor T, the kernel κ_T on the set of data constructors associated with T is default-consistent and constructor-separating. Let $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ be the function defined in Definition 3.7.4. Then k is separating on $\mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.*

Proof. For each $m \in \mathbb{N}$, let $P(m)$ be the property:

> For each $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$ such that $s \neq t$, there exists $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_m$ such that $k(r, s) \neq k(r, t)$.

It is shown by induction that $P(m)$ holds, for all $m \in \mathbb{N}$. The fact that k is separating on each $\mathfrak{B}_\alpha$ follows immediately since, given $s, t \in \mathfrak{B}_\alpha$, there exists an m such that $s, t \in \mathfrak{B}_m$.

First I show that $P(0)$ holds. Let $\alpha \in \mathfrak{S}^c$ and $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_0$ such that $s \neq t$. Thus s and t are nullary data constructors associated with some type constructor T and α has the form $T\ \alpha_1 \ldots \alpha_k$, for some $\alpha_1, \ldots, \alpha_k$ and $k \in \mathbb{N}$. Since κ_T is constructor-separating, there is a nullary data constructor r associated with T such that $\kappa_T(r, s) \neq \kappa_T(r, t)$. Thus $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_0$ and $k(r, s) \neq k(r, t)$. Thus $P(0)$ is proved.

Now assume that $P(m)$ holds. I prove that $P(m+1)$ also holds. There are three cases to consider.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$. Suppose that $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where $s \neq t$. Let s be $C\ s_1 \ldots s_n$ and t be $D\ t_1 \ldots t_m$. There are two cases to consider.
   Suppose first that $C \neq D$. If all the data constructors associated with T are nullary, then there is a nullary data constructor E associated with T such that $\kappa_T(E, C) \neq \kappa_T(E, D)$, since κ_T is constructor-separating. Thus s is C, t is D, $k(E, s) \neq k(E, t)$, and $E \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$. Otherwise, there is a data constructor of arity > 0 and thus κ_T is the discrete kernel. Then $k(s, s) = \kappa_T(C, C) + \sum_{i=1}^{n} k(s_i, s_i) \geq 1$, while $k(s, t) = \kappa_T(C, D) = 0$. Thus $k(s, s) \neq k(s, t)$.
   Suppose now that $C = D$. Thus s is $C\ s_1 \ldots s_n$ and t is $C\ t_1 \ldots t_n$. Let C have signature $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_1 \ldots a_k)$ and $\xi =$

$\{a_1/\alpha_1, \ldots, a_n/\alpha_n\}$. If $k(s,t) \neq k(t,t)$, then $P(m+1)$ is established. Thus it can be assumed that $k(s,t) = k(t,t)$. Since $s \neq t$, there exists $j \in \{1, \ldots, n\}$ such that $s_j \neq t_j$. Furthermore, by Proposition 3.5.5, $s_j, t_j \in \mathfrak{B}_{\sigma_j \xi}$. Now k is a kernel on $\mathfrak{B}_{\sigma_j \xi} \cap \mathfrak{B}_m$, by Proposition 3.7.1, and is separating on this set, by the induction hypothesis. Thus $k(s_j, s_j) - 2k(s_j, t_j) + k(t_j, t_j) \neq 0$, by Proposition 3.7.4, and so $k(s_j, s_j) - k(s_j, t_j) \neq k(s_j, t_j) - k(t_j, t_j)$. Now define $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$ by $r = C\ t_1 \ldots t_{j-1}\ s_j\ t_{j+1} \ldots t_n$. Then $k(s,r) = k(s,t) - k(s_j, t_j) + k(s_j, s_j)$ and $k(t,r) = k(t,t) - k(t_j, t_j) + k(t_j, s_j)$. Hence $k(s,r) \neq k(t,r)$.

2. $\alpha = \beta \to \gamma$. Suppose that $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where $s \neq t$. Since $s \neq t$, there exists $u \in \mathfrak{B}_\beta$ such that $V(s\ u) \neq V(t\ u)$, by Proposition 3.5.10. There are three cases to consider.
   Suppose that $u \in supp(s) \cap supp(t)$. If $k(s,t) \neq k(t,t)$, then $P(m+1)$ is established. Thus it can be assumed that $k(s,t) = k(t,t)$. Denote $V(s\ u)$ by s_u and $V(t\ u)$ by t_u. By Proposition 3.5.6, $s_u, t_u \in \mathfrak{B}_\gamma \cap \mathfrak{B}_m$. Now k is a kernel on $\mathfrak{B}_\gamma \cap \mathfrak{B}_m$, by Proposition 3.7.1, and is separating on this set, by the induction hypothesis. Thus $k(s_u, s_u) - 2k(s_u, t_u) + k(t_u, t_u) \neq 0$, by Proposition 3.7.4, and so $k(s_u, s_u) - k(s_u, t_u) \neq k(s_u, t_u) - k(t_u, t_u)$. Define $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$ to be identical to t except that $V(r\ u) = V(s\ u)$. Then $k(s,r) = k(s,t) - k(s_u, t_u) + k(s_u, s_u)$ and $k(t,r) = k(t,t) - k(t_u, t_u) + k(t_u, s_u)$. Hence $k(s,r) \neq k(t,r)$.
   Suppose that $u \in supp(s)$, but $u \notin supp(t)$. If $k(s,t) \neq k(t,t)$, then $P(m+1)$ is established. Thus it can be assumed that $k(s,t) = k(t,t)$. Now $V(s\ u) \notin \mathfrak{D}$ and so, by Proposition 3.7.5, $k(V(s\ u), V(s\ u)) > 0$. Define $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$ to be identical to t except that $V(r\ u) = V(s\ u)$. Then $k(t,r) = k(t,t)$, since $supp(t) \subset supp(r)$, and $k(s,r) \neq k(s,t)$, since $k(V(s\ u), V(s\ u)) > 0$. Hence $k(s,r) \neq k(t,r)$.
   Suppose that $u \in supp(t)$, but $u \notin supp(s)$. If $k(s,s) \neq k(s,t)$, then $P(m+1)$ is established. Thus it can be assumed that $k(s,s) = k(s,t)$. This case now proceeds analogously to the preceding one with s and t switched.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$. Suppose that $s, t \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$, where $s \neq t$. Let s be $(s_1, \ldots, s_n)$ and t be $(t_1, \ldots, t_n)$. If $k(s,t) \neq k(t,t)$, then $P(m+1)$ is established. Thus it can be assumed that $k(s,t) = k(t,t)$. Now, since $s \neq t$, there exists $j \in \{1, \ldots, n\}$ such that $s_j \neq t_j$. Furthermore, by Proposition 3.5.5, $s_j, t_j \in \mathfrak{B}_{\alpha_j} \cap \mathfrak{B}_m$. Now k is a kernel on $\mathfrak{B}_{\alpha_j} \cap \mathfrak{B}_m$, by Proposition 3.7.1, and is separating on this set, by the induction hypothesis. Thus $k(s_j, s_j) - 2k(s_j, t_j) + k(t_j, t_j) \neq 0$, by Proposition 3.7.4, and so $k(s_j, s_j) - k(s_j, t_j) \neq k(s_j, t_j) - k(t_j, t_j)$. Define $r \in \mathfrak{B}_\alpha \cap \mathfrak{B}_{m+1}$ by $r = (t_1, \ldots, t_{j-1}, s_j, t_{j+1}, \ldots, t_n)$. Then $k(s,r) = k(s,t) - k(s_j, t_j) + k(s_j, s_j)$ and $k(t,r) = k(t,t) - k(t_j, t_j) + k(t_j, s_j)$. Hence $k(s,r) \neq k(t,r)$.

□

The condition in Proposition 3.7.6 that each κ_T be constructor-separating cannot be dropped.

Example 3.7.12. Let κ_{Int} be defined by $\kappa_{Int}(n, m) = 1$, for all $n, m \in \mathbb{Z}$. Then κ_{Int} is default-consistent, but not constructor-separating. Furthermore, the kernel k defined in Definition 3.7.4 using κ_{Int} is not separating (for the integers).

The condition in Proposition 3.7.6 that each κ_T be default-consistent cannot be dropped.

Example 3.7.13. Let κ_{Int} be the product kernel and 1 the default data constructor for the integers. Then κ_{Int} is constructor-separating, but not default-consistent since $\kappa_{Int}(0, 0) = 0$. Let M be a nullary type constructor and $A, B : M$. Then the abstractions $\lambda x.if\ x = A\ then\ 0\ else\ 1$ and $\lambda x.if\ x = B\ then\ 0\ else\ 1$ cannot be separated by another abstraction. Thus the kernel k defined in Definition 3.7.4 using κ_{Int} and the default data constructor 1 for the integers is not separating.

Finally, in this chapter, I return to the issue of defining metrics. Given a kernel, it is easy to define an associated pseudometric.

Proposition 3.7.7. *Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel on $\mathcal{X}$. Then the following hold.*

1. *The function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ defined by*
$$d(x, y) = \sqrt{k(x, x) - 2k(x, y) + k(y, y)},$$
for all $x, y \in \mathcal{X}$, is a pseudometric on $\mathcal{X}$.
2. *If k is separating, then d is a metric.*

Proof. 1. By Proposition 3.7.2, $k(x, y) = \langle \Phi(x), \Phi(y) \rangle$, for all $x, y \in \mathcal{X}$. Thus

$$\begin{aligned} d(x, y) &= \sqrt{k(x, x) - 2k(x, y) + k(y, y)} \\ &= \sqrt{\langle \Phi(x), \Phi(x) \rangle - 2\langle \Phi(x), \Phi(y) \rangle + \langle \Phi(y), \Phi(y) \rangle} \\ &= \sqrt{\langle \Phi(x) - \Phi(y), \Phi(x) - \Phi(y) \rangle} \\ &= \sqrt{\|\Phi(x) - \Phi(y)\|^2} \\ &= \|\Phi(x) - \Phi(y)\|, \end{aligned}$$

where $\|\cdot\|$ is the (canonical) norm on the associated Hilbert space for k.

From this, it is easy to show that $d(x, y) \geq 0$ and $d(x, y) = d(y, x)$, for all $x, y \in \mathcal{X}$. Furthermore, if $x = y$, then $d(x, y) = 0$. For the triangle inequality, note that

$$\begin{aligned} d(x, z) &= \|\Phi(x) - \Phi(z)\| \\ &\leq \|\Phi(x) - \Phi(y)\| + \|\Phi(y) - \Phi(z)\| \\ &= d(x, y) + d(y, z). \end{aligned}$$

Thus d is a pseudometric.

2. Since k is separating, $x \neq y$ implies $k(x, x) - 2k(x, y) + k(y, y) \neq 0$, by Proposition 3.7.4. Thus $x \neq y$ implies $d(x, y) \neq 0$. □

Proposition 3.7.7 provides alternative (pseudo-)metrics on the set of basic terms to those given in Sect. 3.6.

Example 3.7.14. For several choices of k, here is the corresponding metric d given by Proposition 3.7.7.

Let k be the product kernel on a set of numbers, say, $\mathbb{Z}$. Then $d(x,y) = \sqrt{k(x,x) - 2k(x,y) + k(y,y)} = \sqrt{x^2 - 2xy + y^2} = |x-y|$.

More generally, if k is the inner product $\langle \cdot, \cdot \rangle$ on a Hilbert space, then $d(x,y) = \sqrt{\langle x,x \rangle - 2\langle x,y \rangle + \langle y,y \rangle} = \sqrt{\langle x-y, x-y \rangle} = \sqrt{\|x-y\|^2} = \|x-y\|$, the usual metric on the Hilbert space.

Now let δ be the discrete kernel on a set $\mathfrak{X}$. Then $d(x,y) = \sqrt{\delta(x,x) - 2\delta(x,y) + \delta(y,y)} = \sqrt{2 - 2\delta(x,y)}$. Thus $d(x,y) = 0$, if $x = y$; otherwise, $d(x,y) = \sqrt{2}$. Hence d is the discrete metric.

If A and B are (finite) sets and κ_Ω is the discrete kernel, then $d(A,B) = \sqrt{k(A,A) - 2k(A,B) + k(B,B)} = \sqrt{|A| - 2|A \cap B| + |B|} = \sqrt{|A \triangle B|}$, where $\triangle$ is symmetric difference.

Typically, kernel-based learning methods are generic in the sense that they only require that it be possible to define the generalised inner product between pairs of individuals and are independent of the nature of the individuals. Thus the results of this section can be applied directly to a variety of kernel-based learning methods for structured data by simply using the kernels developed here in these algorithms.

Bibliographical Notes

Default, normal, and basic terms first appeared in [55], [56], and [57]. These ideas were originally developed with applications to declarative programming languages [54] and machine learning [7, 57] in mind.

In instance-based learning, a metric is needed to determine those terms that are 'nearby' some given term [61, Chap. 8]. The metric defined in Definition 3.6.3 first appeared in [55] and [56].

For a comprehensive account of kernels and much more about kernel-based learning methods, see [84]. See also [15] and [83]. The details of the proof of Proposition 3.7.2 are given in [84, p.32]. The kernel defined in Definition 3.7.4 first appeared (albeit in a slightly different form) in [29] and [55]. The kernel on sets in Exercise 3.11, Part (iii), was introduced in [33]. Many variants for each component of the definition of the kernel on basic terms are suggested in [84].

There is considerable confusion over the terminology associated with kernels. What is here called a 'kernel' is elsewhere called a 'positive definite kernel', 'Mercer kernel', 'admissible kernel', 'support vector kernel', 'nonnegative definite kernel', or 'covariance kernel' [84, p.30].

Exercises

3.1 Prove or disprove each of the following conjectures.

(i) A subterm of a normal term is normal iff it is closed.
(ii) A subterm of a basic term is basic iff it is closed.

3.2 Prove that $\mathfrak{D} \subseteq \mathfrak{N}$.

3.3 For each $\alpha \in \mathfrak{S}^c$, the set *embed*$(\alpha)$ of *embedded types* in α is defined by *embed*$(\alpha) = \{\beta \mid$ there exists $t \in \mathfrak{N}_\alpha$ and a subterm s of t such that $s \in \mathfrak{N}_\beta\}$.

(i) Let T_1 and T_2 be nullary type constructors. Suppose there is a data constructor having signature $T_1 \times T_1 \rightarrow T_1 \times T_1 \rightarrow T_2$, and there are data constructors having signature T_1. Consider the type $\alpha = T_2 \times \{Int\} \times$ *List Float*. Determine *embed*(α).
(ii) Prove or disprove: *embed*(α) is the set of subtypes of α.

3.4 The binary relation $\lhd$ on $\mathfrak{N}$ is defined inductively as follows. Let $s, t \in \mathfrak{N}$. Then $s \lhd t$ if there exists $\alpha \in \mathfrak{S}^c$ such that $s, t \in \mathfrak{N}_\alpha$ and one of the following conditions holds.

1. $\alpha = T\ \alpha_1 \ldots \alpha_k$, for some $T, \alpha_1, \ldots, \alpha_k$, and s is $C\ s_1 \ldots s_n$, t is $D\ t_1 \ldots t_m$ and either $C \prec_T D$ or $C = D$ and there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$.
2. $\alpha = \beta \rightarrow \gamma$, for some β, γ, and either $V(s\ r) \equiv V(t\ r)$ or $V(s\ r) \lhd V(t\ r)$, for all $r \in \mathfrak{N}_\beta$, and $V(s\ b) \lhd V(t\ b)$, for some $b \in \mathfrak{N}_\beta$.
3. $\alpha = \alpha_1 \times \cdots \times \alpha_n$, for some $\alpha_1, \ldots, \alpha_n$, and s is $(s_1, \ldots, s_n)$, t is $(t_1, \ldots, t_n)$ and there exists j such that $s_1 \equiv t_1, \ldots, s_{j-1} \equiv t_{j-1}$, $s_j \lhd t_j$ and $1 \leq j \leq n$.

Prove the following.

(i) For each $\alpha \in \mathfrak{S}^c$, if $s, t \in \mathfrak{N}_\alpha$ and $s \equiv t$, then $s \not\lhd t$.
(ii) For each $\alpha \in \mathfrak{S}^c$, if $r, s, t \in \mathfrak{N}_\alpha$, $r \equiv s$ and $s \lhd t$, then $r \lhd t$.
(iii) For each $\alpha \in \mathfrak{S}^c$, if $r, s, t \in \mathfrak{N}_\alpha$, $r \lhd s$ and $s \equiv t$, then $r \lhd t$.
(iv) For each $\alpha \in \mathfrak{S}^c$, $\lhd|_{\mathfrak{N}_\alpha}$ is a strict partial order on $\mathfrak{N}_\alpha$.
(v) Assuming that $\bot \prec_\Omega \top$, the partial order $\lhd$ on sets corresponds to strict set inclusion, that is, $s \lhd t$ iff $s \subset t$.
(vi) Assuming that $\prec_{Int}$ is the usual $<$ on the integers, the partial order $\lhd$ on multisets corresponds to strict multiset inclusion, that is, $s \lhd t$ iff $s \sqsubset t$.

3.5 The binary relation $\unlhd$ on $\mathfrak{N}$ is defined by $s \unlhd t$ if $s \lhd t$ or $s \equiv t$. Prove the following.

(i) For each $\alpha \in \mathfrak{S}^c$, $\unlhd|_{\mathfrak{N}_\alpha}$ is a preorder on $\mathfrak{N}_\alpha$.
(ii) $\unlhd$ is not a partial order on $\mathfrak{N}_\alpha$, since it is not antisymmetric.
(iii) For each $\alpha \in \mathfrak{S}^c$, if $s, t \in \mathfrak{N}_\alpha$, then $s \equiv t$ iff $s \unlhd t$ and $t \unlhd s$.
(iv) For each $\alpha \in \mathfrak{S}^c$, $\unlhd|_{\mathfrak{B}_\alpha}$ is a partial order on $\mathfrak{B}_\alpha$.

(v) The preorder $\trianglelefteq$ on sets corresponds to set inclusion, that is, $s \trianglelefteq t$ iff $s \subseteq t$.
(vi) The preorder $\trianglelefteq$ on multisets corresponds to multiset inclusion, that is, $s \trianglelefteq t$ iff $s \sqsubseteq t$.

3.6 Let φ be a non-decreasing function from the non-negative reals into the closed interval $[0, 1]$ such that $\varphi(0) = 0$, $\varphi(x) > 0$ if $x > 0$, and $\varphi(x + y) \leq \varphi(x) + \varphi(y)$, for each x and y. Let $(\mathfrak{X}, d)$ be a metric space. Prove that the function d' defined by $d'(x, y) = \varphi(d(x, y))$, for all $x, y \in \mathfrak{X}$, is a bounded metric on $\mathfrak{X}$.

3.7

(i) Suppose that the set of data constructors is countable. Prove that, for each $\alpha \in \mathfrak{S}^c$, the metric space $(\mathfrak{B}_\alpha, d)$ of Definition 3.6.3 is separable.
(ii) Suppose that $\mathfrak{B}_\alpha$ is uncountable, for some $\alpha_0 \in \mathfrak{S}^c$. Prove that the metric space $(\mathfrak{B}_{\alpha_0 \to \Omega}, d)$ of Definition 3.6.3 is not separable.

3.8 Let $\mathfrak{X}$ be a set and $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ a function. Prove that k positive definite on $\mathfrak{X}$ iff, for all $n \in \mathbb{Z}^+, x_1, \ldots, x_n \in \mathfrak{X}$, where $x_i \neq x_j$, for $i \neq j$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that $\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) \geq 0$.

3.9 Let $\mathfrak{X}$ be a set and $f : \mathfrak{X} \to \mathbb{R}$ a function. Prove that $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ defined by $k(x, y) = f(x) f(y)$ is positive definite.

3.10 Let $k : \mathbb{R}^n \to \mathbb{R}$ be defined by $k(x, y) = \exp(-\|x - y\|^2/\sigma^2)$, where $\|\cdot\|$ is the Euclidean norm on $\mathbb{R}^n$ and $\sigma > 0$. Prove that k is positive definite. [Hint: $\exp(-\|x - y\|^2/\sigma^2) = \exp(-\|x\|^2/\sigma^2) \exp(-\|y\|^2/\sigma^2) \exp(2\langle x, y\rangle/\sigma^2)$.]

3.11 The function $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively on the structure of terms in $\mathfrak{B}$ as follows. Let $s, t \in \mathfrak{B}$. Parts 1, 3, and 4 are the same as for Definition 3.7.4. Part 2 is as follows.

If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then

$$k(s, t) = \sum_{\substack{u \in supp(s) \\ v \in supp(t)}} k(V(s\ u), V(t\ v)) \cdot k(u, v).$$

(i) Prove that k is a kernel on $\mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.
(ii) Suppose that k is the discrete kernel on $\mathfrak{B}_\beta$. Prove that

$$k(s, t) = \sum_{u \in supp(s) \cap supp(t)} k(V(s\ u), V(t\ u)),$$

for $s, t \in \mathfrak{B}_{\beta \to \gamma}$.
(iii) Suppose that γ is Ω and k is the discrete kernel on $\mathfrak{B}_\gamma$. Prove that

$$k(s, t) = \sum_{\substack{u \in supp(s) \\ v \in supp(t)}} k(u, v),$$

for $s, t \in \mathfrak{B}_{\beta \to \gamma}$.

3.12 Show that the kernel k on sets of integers defined by

$$k(A, B) = \sum_{u \in A, v \in B} uv$$

is not separating (and thus the kernel of Exercise 3.11 is not separating, in general).

3.13 The function $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ is defined inductively on the structure of terms in $\mathfrak{B}$ as follows. Let $s, t \in \mathfrak{B}$. Parts 1, 3, and 4 are the same as for Definition 3.7.4. Part 2 is as follows.

If $s, t \in \mathfrak{B}_\alpha$, where $\alpha = \beta \to \gamma$, for some β, γ, then

$$k(s, t) = \sum_{\substack{u \in supp(s) \\ v \in supp(t)}} k(V(s\ u), V(s\ u)) \cdot k(V(t\ v), V(t\ v)) \cdot k(u, v).$$

Prove that k is a kernel on $\mathfrak{B}_\alpha$, for each $\alpha \in \mathfrak{S}^c$.

3.14 Let $\mathfrak{X}$ be a set. A function $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ is *strictly positive definite* on $\mathfrak{X}$ if, for all $n \in \mathbb{Z}^+, x_1, \ldots, x_n \in \mathfrak{X}$, and $c_1, \ldots, c_n \in \mathbb{R}$, it follows that

1. $\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) \geq 0$, and
2. $\sum_{i,j \in \{1,\ldots,n\}} c_i\, c_j\, k(x_i, x_j) = 0$ implies $c_i = 0$, for $i = 1, \ldots, n$.

A function $k : \mathfrak{X} \times \mathfrak{X} \to \mathbb{R}$ is a *strict kernel* if it is symmetric and strictly positive definite.

(i) Give an example to show that the product kernel on $\mathbb{R}$ is not a strict kernel.
(ii) Prove that if k is a strict kernel, then k is separating.

4. Predicates

Having established the use of the logic for representing individuals, attention now turns to the problem of constructing predicates that individuals may or may not satisfy. Essentially, all that is required is the definition of a suitable collection of predicates on the type of an individual. However, these predicates are usually built up incrementally by composition and it is this incremental construction that is studied here.

4.1 Transformations

Composition is handled by the (reverse) composition function

$$\circ : (a \to b) \to (b \to c) \to (a \to c)$$

defined by

$$((f \circ g)\ x) = (g\ (f\ x)).$$

Composition is right associative. Thus the expression $f_1 \circ \cdots \circ f_n$ means $f_1 \circ (f_2 \circ \cdots \circ (f_{n-1} \circ f_n) \cdots)$. More precisely, using the usual notation for application, this expression is $((\circ\, f_1)\ ((\circ\, f_2) \cdots ((\circ\, f_{n-1})\ f_n) \cdots)$.

Predicates are built up by composing transformations, which are defined as follows.

Definition 4.1.1. A *transformation* f is a function having a signature of the form

$$f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \mu \to \sigma,$$

where any parameters in $\varrho_1, \ldots, \varrho_k$ and σ appear in μ, and $k \geq 0$. The type μ is distinguished and is called the *source* of the transformation, while the type σ is called the *target* of the transformation. The number k is called the *rank* of the transformation.

Since juxtaposition is left associative, the expression $f\ p_1 \ldots p_k$ means $(\cdots(((f\ p_1)\ p_2)\ p_3) \ldots p_k)$. The intuitive idea behind the definition of transformation is that, given predicates $p_i : \varrho_i \to \Omega$, for $i = 1, \ldots, k$, $f\ p_1 \ldots p_k$

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

is a function that takes individuals of type μ to individuals of type σ. By composing (generally) several such functions, the last of which is a predicate, a predicate on individuals of the desired type is obtained.

Note that every function having signature $\mu \to \sigma$ is potentially a transformation. (Just put $k = 0$.) It is understood that some collection of functions having appropriate signatures are *declared* to be transformations for each particular application.

The remainder of this section provides useful transformations for the various types.

Example 4.1.1. The transformation

$$\wedge_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to a \to \Omega$$

defined by

$$\wedge_n \; p_1 \ldots p_n \; x = (p_1 \; x) \wedge \cdots \wedge (p_n \; x),$$

where $n \geq 2$, provides a 'conjunction' with n conjuncts. Transformations analogous to the other connectives can be defined similarly.

Example 4.1.2. Following the style of Example 4.1.1, the transformation

$$\sim : (a \to \Omega) \to a \to \Omega$$

defined by

$$\sim \; p \; x = \neg(p \; x),$$

provides negation. But negation can also be introduced directly by the transformation

$$\neg : \Omega \to \Omega$$

which is the usual negation connective. Then the transformation $(\sim \; p)$ can be written equivalently as $p \circ \neg$. (In practice, just one of these methods of introducing negation would be used.)

Example 4.1.3. Each projection

$$proj_i : a_1 \times \cdots \times a_n \to a_i$$

defined by

$$proj_i \; (t_1, \ldots, t_n) = t_i,$$

for $i = 1, \ldots, n$, is a transformation of rank 0.

Example 4.1.4. There are two fundamental transformations $top : a \rightarrow \Omega$ and $bottom : a \rightarrow \Omega$ defined by $top\ x = \top$ and $bottom\ x = \bot$, for each x. Each of *top* and *bottom* is a constant predicate, with *top* being the weakest predicate on the type a and *bottom* being the strongest.

Example 4.1.5. Let μ be a type and suppose that $A : \mu$, $B : \mu$, and $C : \mu$ are constants. Then, corresponding to A, one can define a transformation

$$(= A) : \mu \rightarrow \Omega$$

by

$$((= A)\ x) = x = A,$$

with analogous definitions for $(= B)$ and $(= C)$. Similarly, one can define the transformation

$$(\neq A) : \mu \rightarrow \Omega$$

by

$$((\neq A)\ x) = x \neq A.$$

Example 4.1.6. Consider a type such as *Int* which has various order relations defined on it. Then, for any integer N, one can define the transformation

$$(< N) : Int \rightarrow \Omega$$

by

$$((< N)\ m) = m < N.$$

In a similar way, one can define the transformations $(> N)$, $(\geq N)$, and $(\leq N)$.

Example 4.1.7. The if-then-else transformation

$$ite : (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow (a \rightarrow \Omega) \rightarrow a \rightarrow \Omega$$

is defined by

$$ite\ p\ q\ r\ x = if\ (p\ x)\ then\ (q\ x)\ else\ (r\ x).$$

As an illustration, one could define the predicate

$$ite\ (\sim null)\ (head \circ (= 42))\ bottom,$$

where *null* checks whether a list is empty or not and *head* extracts the head of a non-empty list. This predicate returns $\top$ iff its argument is a non-empty list with 42 as its first item.

Example 4.1.8. Consider again the type *Shape* and the data constructors

$$Circle : Float \rightarrow Shape$$
$$Rectangle : Float \rightarrow Float \rightarrow Shape.$$

The function $isCircle : (Float \rightarrow \Omega) \rightarrow Shape \rightarrow \Omega$ defined by

$$isCircle\ p\ t = \exists x.((t = Circle\ x) \wedge (p\ x))$$

is a transformation. Similarly, the function $isRectangle : (Float \rightarrow \Omega) \rightarrow (Float \rightarrow \Omega) \rightarrow Shape \rightarrow \Omega$ defined by

$$isRectangle\ p\ q\ t = \exists x.\exists y.((t = Rectangle\ x\ y) \wedge (p\ x) \wedge (q\ y))$$

is a transformation. For predicates p and q, $(isRectangle\ p\ q)$ is a predicate on geometrical shapes which returns $\top$ iff the shape is a rectangle whose length satisfies p and whose breadth satisfies q.

Before giving further examples, a reminder on notation. Sets and predicates have been identified, so that a set has type $\mu \rightarrow \Omega$, for some type μ. However, in practice, even though sets and predicates have been identified, for a term of type $\mu \rightarrow \Omega$, it is sometimes convenient to make an informal distinction depending upon whether one is thinking of the term as a 'set of elements' or as a 'condition'. In the former case, the synonym $\{\mu\}$ for $\mu \rightarrow \Omega$ is used to indicate this. To emphasise, there is no mathematical distinction between $\{\mu\}$ and $\mu \rightarrow \Omega$, but there is a change in the intuitive role the corresponding term is understood to be playing. Similarly, the notation '$s \in t$' is used if one is thinking of t as a 'set of elements' and the notation '$(t\ s)$' is used if one is thinking of t as a 'condition'. This distinction will be very convenient in the following discussion.

Example 4.1.9. Consider the transformation

$$domCard : (a \rightarrow \Omega) \rightarrow \{a\} \rightarrow Nat$$

defined by

$$domCard\ p\ t = card\ \{x \mid (p\ x) \wedge x \in t\},$$

where *card* computes the cardinality of a (finite) set. Given a predicate p on some type μ and a unary predicate on *Nat* such as (> 42), which returns $\top$ iff its argument is strictly greater than 42, one can construct a predicate $(domCard\ p) \circ (> 42)$ on sets of type $\{\mu\}$ which selects the subset of elements that satisfy the predicate p and then checks that the cardinality of this subset is greater than 42.

Example 4.1.10. Consider the transformation

$$setExists_1 : (a \to \Omega) \to \{a\} \to \Omega$$

defined by

$$setExists_1\ p\ t = \exists x.((p\ x) \wedge (x \in t)).$$

The predicate ($setExists_1\ p$) checks whether a set has an element that satisfies p. More generally, for $n \geq 1$, one can define

$$setExists_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to \{a\} \to \Omega$$

by

$$\begin{aligned} setExists_n\ p_1 \ldots p_n\ t = \exists x_1. \cdots \exists x_n.\ & ((p_1\ x_1) \wedge \cdots \wedge (p_n\ x_n) \wedge \\ & (x_1 \in t) \wedge \cdots \wedge (x_n \in t) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n)). \end{aligned}$$

Note the overlap between $(domCard\ p) \circ (> 0)$ and $(setExists_1\ p)$. Typically, $setExists_n$ is used for small values of n, say, 1, 2 or 3, while *domCard* is used in conjunction with $(> N)$ for larger values of N.

Example 4.1.11. Multisets (also known as bags) are a useful generalisation of sets. A straightforward approach to multisets is to regard a multiset as a function of type $\mu \to Nat$, where μ is the type of elements in the multiset and the value of the multiset on some item is its multiplicity, that is, the number of times it occurs in the multiset. (In fact, a set can be identified with a multiset taking only the values 0 and 1, and so sets are redundant. However, sets occur more commonly than multisets and are more familiar, thus I prefer to keep both.) Consider the transformation

$$\begin{aligned} & domMcard : (a \to \Omega) \to (a \to Nat) \to Nat \\ & domMcard\ p\ t = mcard\ (\lambda x.if\ (p\ x)\ then\ (t\ x)\ else\ 0), \end{aligned}$$

where *mcard* computes the cardinality of a multiset. Thus, for each $p : \mu \to \Omega$,

$$(domMcard\ p) : (\mu \to Nat) \to Nat$$

is a function on multisets whose elements have type μ that computes the cardinality of the submultiset of elements of the argument that satisfy the predicate p.

One can then obtain a predicate by composing $(domMcard\ p)$ with, say, a predicate $(> N)$, for some $N \geq 0$. This gives the predicate $(domMcard\ p) \circ (> N)$, which is true of a multiset iff the cardinality of the submultiset of elements satisfying the predicate p is strictly greater than N.

Multisets also support the analogue of the transformations $setExists_n$ for sets. The transformation $msetExists_n$, for $n \geq 1$, is defined as follows.

$$msetExists_n : (a \to \Omega) \to \cdots \to (a \to \Omega) \to (a \to Nat) \to \Omega$$
$$msetExists_n\ p_1 \ldots p_n\ t = \exists x_1. \cdots \exists x_n.((p_1\ x_1) \wedge \cdots \wedge (p_n\ x_n) \wedge$$
$$((t\ x_1) > 0)\ \wedge \cdots \wedge\ ((t\ x_n) > 0) \wedge (x_1 \neq x_2) \wedge \cdots \wedge (x_{n-1} \neq x_n)).$$

Example 4.1.12. Lists are represented using the following data constructors.

$$[] : List\ a$$
$$\sharp : a \to List\ a \to List\ a.$$

A list has a type $List\ \mu$, where μ is the type of the items in the list. Lists are constructed as usual from the empty list [] and the cons function $\sharp$.

The most basic transformations on lists that are useful for inductive learning are *null*, which tests whether a list is empty or not; *head* which returns the first item in a list; *tail* which returns the tail of a list; and *last* which returns the last item in a list. Given a non-negative integer N as an index and a list, the transformation $(!!N)$ returns the item at that index in the list. (The index of the first item is 0.) If the order of items in a list is not important, the list can be converted to a multiset with the transformation *listToMultiset* and then predicates can be applied to the multiset. The transformation *listToSet* returns the set of items in a list. One can also pull out sublists from the list: given an integer $N \geq 2$, (*sublists* N) is a transformation that returns the set of sublists of contiguous items of length N. Finally, *filterLength* takes a predicate and a list as arguments and returns the length of the list obtained from the original one by including only those items that satisfy the predicate. Here are the signatures of these transformations.

$$null : List\ a \to \Omega$$
$$head : List\ a \to a$$
$$tail : List\ a \to List\ a$$
$$last : List\ a \to a$$
$$(!!N) : List\ a \to a$$
$$listToMultiset : List\ a \to (a \to Nat)$$
$$listToSet : List\ a \to \{a\}$$
$$(sublists\ N) : List\ a \to \{List\ a\}$$
$$filterLength : (a \to \Omega) \to List\ a \to Nat.$$

Example 4.1.13. Next I consider binary trees, the standard representation of which uses the following data constructors.

$$Null : BTree\ a$$
$$BNode :\ BTree\ a \to a \to BTree\ a \to BTree\ a.$$

$BTree\ \mu$ is the type of a binary tree whose nodes have type μ, *Null* represents the empty binary tree and *BNode* is a data constructor whose first argument

is the left subtree, the second is the root node, and the third is the right subtree of the tree.

Some useful transformations on binary trees for making predicates are *emptyBTree*, which tests whether a tree is empty or not; *root*, which returns the root node; *leftTree*, which returns the left subtree; *rightTree*, which returns the right subtree; *nodeSet*, which returns the set of nodes in the tree; *nodeMultiset*, which returns the multiset of nodes in the tree; and *branches*, which returns the list of branches (ordered from left to right), where each branch is regarded as a list of nodes beginning with the root and ending with a leaf node. The signatures for these transformations are as follows.

$$
\begin{aligned}
&emptyTree : BTree\ a \rightarrow \Omega \\
&root : BTree\ a \rightarrow a \\
&leftTree : BTree\ a \rightarrow BTree\ a \\
&rightTree : BTree\ a \rightarrow BTree\ a \\
&nodeSet : BTree\ a \rightarrow \{a\} \\
&nodeMultiset : BTree\ a \rightarrow (a \rightarrow Nat) \\
&branches : BTree\ a \rightarrow List\ (List\ a).
\end{aligned}
$$

Example 4.1.14. The standard representation of a tree (where the subtrees have an order) uses the data constructor *Node*, where

$$Node : a \rightarrow List\ (Tree\ a) \rightarrow Tree\ a.$$

Tree μ is the type of a tree and *Node* is a data constructor whose first argument is the root node and second argument is the list of subtrees.

Some useful transformations on trees for making predicates are *root*, which returns the root node; *numberOfSubtrees*, which returns the number of subtrees; (*subtree* N), which returns the Nth subtree; *nodeSet*, which returns the set of all nodes in the tree; *nodeMultiset*, which returns the multiset of all nodes in the tree; and *branches*, which returns the list of branches (in the order induced by the order of the subtrees).

$$
\begin{aligned}
&root : Tree\ a \rightarrow a \\
&numberOfSubtrees : Tree\ a \rightarrow Nat \\
&(subtree\ N) : Tree\ a \rightarrow Tree\ a \\
&nodeSet : Tree\ a \rightarrow \{a\} \\
&nodeMultiset : Tree\ a \rightarrow (a \rightarrow Nat) \\
&branches : Tree\ a \rightarrow List\ (List\ a).
\end{aligned}
$$

Example 4.1.15. Suppose that *Graph* $\nu\ \varepsilon$ is the type of (undirected) graphs whose vertices have type *Vertex* $\nu\ \varepsilon$ and edges have type *Edge* $\nu\ \varepsilon$, where ν is the type of the information at a vertex and ε is the type of information

(for example, the weight) on an edge. Consider the following transformations on graphs.

$$
\begin{aligned}
&\mathit{vertices} : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Vertex}\ \nu\ \varepsilon\}\\
&\mathit{edges} : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Edge}\ \nu\ \varepsilon\}\\
&\mathit{vertex} : \mathit{Vertex}\ \nu\ \varepsilon \to\ \nu\\
&\mathit{edge} : \mathit{Edge}\ \nu\ \varepsilon \to\ \varepsilon\\
&\mathit{connects} : \mathit{Edge}\ \nu\ \varepsilon \to (\mathit{Vertex}\ \nu\ \varepsilon \to \mathit{Nat})\\
&\mathit{edgesAtVertex} : \mathit{Vertex}\ \nu\ \varepsilon \to \{\mathit{Edge}\ \nu\ \varepsilon\}\\
&(\mathit{subgraphs}\ N) : \mathit{Graph}\ \nu\ \varepsilon \to \{\mathit{Graph}\ \nu\ \varepsilon\}.
\end{aligned}
$$

Here *vertices* returns the set of vertices of a graph, *edges* returns the set of edges of a graph, *vertex* returns the information at a vertex, *edge* returns the information on an edge, *connects* returns the unordered pair of vertices joined by an edge, *edgesAtVertex* returns the set of edges ending at a vertex, and (*subgraphs* N) returns the set of (connected) subgraphs containing N vertices of a graph.

With these transformations, the predicate

$$\mathit{vertices} \circ (\mathit{domCard}\ (\mathit{vertex} \circ (\wedge_2\ p\ q))) \circ (> 7)$$

on individuals of type $\mathit{Graph}\ \nu\ \varepsilon$ can be constructed, where p and q are predicates on individuals of type ν. This predicate is true for a graph iff the cardinality of the subset of vertices in the graph whose information satisfies the predicates p and q is greater than 7.

One can also construct the predicate

$$(\mathit{subgraphs}\ 3) \circ (\mathit{domCard}\ (\mathit{edges} \circ (\mathit{setExists}_2\ (\mathit{edge} \circ p)\ (\mathit{edge} \circ q)))) \circ (> 3),$$

where p and q are predicates on the type ε. This predicate is true for a graph iff the cardinality of the subset of (connected) subgraphs in the graph that have three vertices and two (distinct) edges, one of which has information satisfying p and one satisfying q, is greater than 3.

Example 4.1.16. The type of a directed graph is $\mathit{Digraph}\ \nu\ \varepsilon$, where ν is the type of information in the vertices and ε is the type of information in the edges, and

$$\mathit{Digraph}\ \nu\ \varepsilon = \{\mathit{Label} \times \nu\} \times \{(\mathit{Label} \times \mathit{Label}) \times \varepsilon\}.$$

There is a type constructor *DiVertex* such that the type of a vertex is $\mathit{DiVertex}\ \nu\ \varepsilon$ and a type constructor *DiEdge* such that the type of an edge is $\mathit{DiEdge}\ \nu\ \varepsilon$. There are also the following transformations.

$$
\begin{aligned}
&vertices : Digraph\ \nu\ \varepsilon \to \{DiVertex\ \nu\ \varepsilon\}\\
&edges : Digraph\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}\\
&vertex : DiVertex\ \nu\ \varepsilon \to \nu\\
&edge : DiEdge\ \nu\ \varepsilon \to \varepsilon\\
&connects : DiEdge\ \nu\ \varepsilon \to (DiVertex\ \nu\ \varepsilon) \times (DiVertex\ \nu\ \varepsilon)\\
&edgesToVertex : DiVertex\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}\\
&edgesFromVertex : DiVertex\ \nu\ \varepsilon \to \{DiEdge\ \nu\ \varepsilon\}\\
&(subgraphs\ N) : Digraph\ \nu\ \varepsilon \to \{Digraph\ \nu\ \varepsilon\}.
\end{aligned}
$$

The transformation *vertices* returns the set of vertices of a digraph, *edges* returns the set of edges of a digraph, *vertex* returns the information at a vertex, *edge* returns the information on an edge, *connects* returns the (ordered) pair of vertices joined by an edge, *edgesToVertex* returns the set of edges that are directed into some vertex, *edgesFromVertex* returns the set of edges that are directed away from some vertex, and (*subgraphs* N) returns the set of (connected) subdigraphs of the digraph that contain N vertices.

I have concentrated above on generic transformations; that is, given some type, I have provided a set of transformations that is likely to be generally useful in applications involving that type. I do not claim to have given *all* generic transformations for each type, just useful collections so that any particular application will probably only need a subset of the ones I provide. However, it is common in applications to also have need of domain-specific transformations that capture some specialised concepts that will be useful in the application. For example, in predictive toxicology applications a number of domain-specific predicates have been studied, including such things as the possible existence of 2-dimensional substructures like benzene rings and methyl groups. Such predicates are not relevant in general graphs, but they can be important if the graphs are a representation of molecules. Technically, such domain-specific transformations are handled in exactly the same way as generic transformations are handled in this book. Domain-specific transformations are added to the hypothesis language. Their definitions become part of the background theory and can be used in the construction of predicates in the same way as generic transformations.

4.2 Standard Predicates

Next the definition of the class of predicates formed by composing transformations is presented. First, a larger class of predicates, the standard predicates, is defined and then the desired subclass, the regular predicates, which has much less redundancy, is defined. In the following definition, it is assumed that some (possibly infinite) class of transformations is given and all transformations considered are taken from this class. A standard predicate is defined

by induction on the number of (occurrences of) transformations it contains as follows.

Definition 4.2.1. A *standard predicate* is a term of the form

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}),$$

where f_i is a transformation of rank k_i $(i = 1, \ldots, n)$, the target of f_n is Ω, p_{i,j_i} is a standard predicate $(i = 1, \ldots, n, \quad j_i = 1, \ldots, k_i)$, $k_i \geq 0$ $(i = 1, \ldots, n)$ and $n \geq 1$.

The set of all standard predicates is denoted by $\mathbf{S}$. Standard predicates have type of the form $\mu \to \Omega$, for some type μ. Note that the set of standard predicates is defined *relative to* a previously declared set of transformations.

Definition 4.2.2. For each $\alpha \in \mathfrak{S}^c$, define $\mathbf{S}_\alpha = \{p \in \mathbf{S} \mid p$ has type $\mu \to \Omega$ and μ is more general than $\alpha\}$.

The intuitive meaning of $\mathbf{S}_\alpha$ is that it is the set of all predicates of a particular form given by the transformations on individuals of type α.

Careful attention needs to be paid to the subterms of a standard predicate, especially the subterms that are themselves standard predicates.

Definition 4.2.3. Let $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$ be a standard predicate. A *suffix* of the standard predicate is a term of the form

$$(f_i\ p_{i,1} \ldots p_{i,k_i}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}),$$

for some $i \in \{1, \ldots, n\}$. The suffix is *proper* if $i > 1$.

A *prefix* of the standard predicate is a term of the form

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_i\ p_{i,1} \ldots p_{i,k_i}),$$

for some $i \in \{1, \ldots, n\}$. The prefix is *proper* if $i < n$.

A suffix of a standard predicate is a standard predicate, but a prefix of a standard predicate may not be a predicate.

Proposition 4.2.1. *Let p be a standard predicate $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$ and q a term. Then q is a subterm of p iff (at least) one of the following conditions holds.*

1. *q is a suffix of p.*
2. *q is a subterm of $(\circ\,(f_i\ p_{i,1} \ldots p_{i,k_i}))$, for some $i \in \{1, \ldots, n-1\}$.*
3. *q is a subterm of $(f_n\ p_{n,1} \ldots p_{n,k_n})$,*

Proof. The proof is by induction on n. If $n = 1$, the result is obvious. Consider now a standard predicate

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_{n+1}\ p_{n+1,1} \ldots p_{n+1,k_{n+1}}),$$

which can be written in the form

$$((\circ\,(f_1\ p_{1,1} \ldots p_{1,k_1}))\ \ (f_2\ p_{2,1} \ldots p_{2,k_2}) \circ \cdots \circ (f_{n+1}\ p_{n+1,1} \ldots p_{n+1,k_{n+1}})).$$

A subterm of this is either the term itself or a subterm of $(\circ\,(f_1\ p_{1,1} \ldots p_{1,k_1}))$ or a subterm of $(f_2\ p_{2,1} \ldots p_{2,k_2}) \circ \cdots \circ (f_{n+1}\ p_{n+1,1} \ldots p_{n+1,k_{n+1}})$. The result follows by applying the induction hypothesis to this last term. □

Proposition 4.2.2. *Let p be a standard predicate $(f_1\ \ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ \ p_{n,1} \ldots p_{n,k_n})$ and q a subterm of p. Then q is a standard predicate iff (at least) one of the following conditions holds.*

1. *q is a suffix of p.*
2. *q has the form $(f_i\ p_{i,1} \ldots p_{i,k_i})$, for some $i \in \{1, \ldots, n\}$, and f_i has target Ω.*
3. *q is a subterm of p_{i,j_i}, for some i and j_i, and q is a standard predicate.*

Proof. According to Proposition 4.2.1, the subterms of p are the suffixes of p, the subterms of $(\circ\,(f_i\ \ p_{i,1} \ldots p_{i,k_i}))$, for $i \in \{1, \ldots, n-1\}$, and the subterms of $(f_n\ p_{n,1} \ldots p_{n,k_n})$, Thus it is only necessary to establish which of these subterms are standard predicates. First, all the suffixes are standard predicates. Since $\circ$ is not a standard predicate, it remains to investigate the subterms of $(f_i\ p_{i,1} \ldots p_{i,k_i})$, for $i = 1, \ldots, n$. Now, provided the target of f_i is Ω, $(f_i\ p_{i,1} \ldots p_{i,k_i})$ is a standard predicate. Also the only proper subterms of $(f_i\ p_{i,1} \ldots p_{i,k_i})$ that could possibly be standard predicates are subterms of $p_{i,1}, \ldots, p_{i,k_i}$. (The reason is that for f_i to be the first symbol in a standard predicate, it must be followed by *all* k of its predicate arguments.) The result follows. □

Example 4.2.1. Consider the standard predicate

$$(f_1\ p) \circ (f_2\ q\ r) \circ (f_3\ s),$$

where the transformations f_1, f_2, f_3, p, q, r, and s have suitable signatures. Thus p, q, r, and s are predicates. Suppose also that neither f_1 nor f_2 have target Ω. Taking the associativity into account, this standard predicate can be written more precisely as

$$((\circ\,(f_1\ p))\ \ ((\circ\,((f_2\ q)\ r))\ \ (f_3\ s))).$$

The subterms of $(f_1\ p) \circ (f_2\ q\ r) \circ (f_3\ s)$ are illustrated in Fig. 4.1. The proper subterms that are standard predicates are $(f_2\ q\ r) \circ (f_3\ s)$, $(f_3\ s)$, p, q, r, and s.

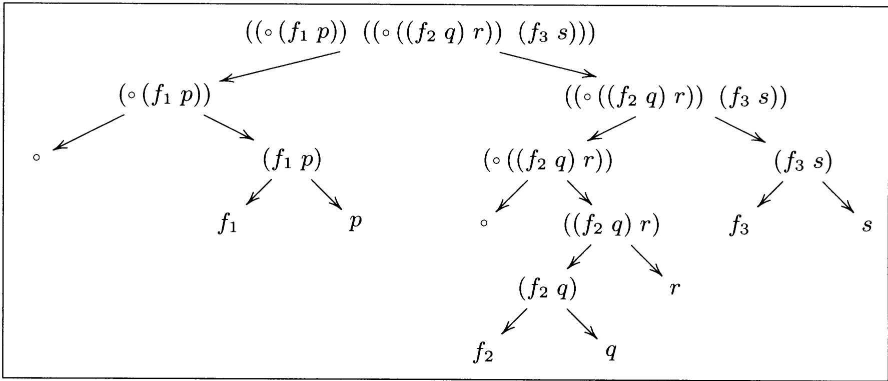

Fig. 4.1. Subterms of a standard predicate

Next an important class of subterms for later use in predicate rewrite systems is introduced.

Definition 4.2.4. A subterm of a standard predicate $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$ is *eligible* if it is a suffix of the standard predicate or it is an eligible subterm of p_{i,j_i}, for some $i \in \{1, \dots, n\}$ and $j_i \in \{1, \dots, k_i\}$.

Example 4.2.2. The eligible subterms of

$$vertices \circ (setExists_2\ (\wedge_2\ (proj_1 \circ (= A))\ (proj_2 \circ (= 3)))\ (proj_1 \circ (= B)))$$

are

$$vertices \circ (setExists_2\ (\wedge_2\ (proj_1 \circ (= A))\ (proj_2 \circ (= 3)))\ (proj_1 \circ (= B))),$$
$$setExists_2\ (\wedge_2\ (proj_1 \circ (= A))\ (proj_2 \circ (= 3)))\ (proj_1 \circ (= B)),$$
$$\wedge_2\ (proj_1 \circ (= A))\ (proj_2 \circ (= 3)),$$
$$proj_1 \circ (= A),$$
$$proj_2 \circ (= 3),$$
$$(= A),$$
$$(= 3),$$
$$proj_1 \circ (= B), \text{ and}$$
$$(= B).$$

Proposition 4.2.3. *An eligible subterm of a standard predicate is a standard predicate.*

Proof. This is a straightforward induction argument. □

Proposition 4.2.4. *Let p, q, and r be standard predicates, and q an eligible subterm of p such that $p[q/r]$ is a term. Then $p[q/r]$ is a standard predicate.*

Proof. The proof is by induction on the number of transformations in p. Suppose the result holds for standard predicates that have $< m$ transformations and p has m transformations. Let p have the form $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$. If q is a suffix of p, then the result follows since $p[q/r]$ is a term. If q is an eligible subterm of some p_{i,j_i}, then the induction hypothesis gives that $p_{i,j_i}[q/r]$ is a standard predicate. Since $p[q/r]$ is a term, it follows that $p[q/r]$ is a standard predicate. □

The class of standard predicates just defined contains some redundancy in that there are syntactically distinct predicates that are equivalent, in a certain sense.

Definition 4.2.5. The theory consisting of the definitions of the transformations (and associated functions) is called the *background theory.*

Definition 4.2.6. Let $\mathcal{B}$ be the background theory, and s and t be terms. Then s and t are *equivalent with respect to* $\mathcal{B}$ if the types of s and t are the same (up to variants) and $s = t$ is a logical consequence of $\mathcal{B}$.

Equivalence with respect to a background theory is an equivalence relation on the set of terms. When discussing equivalence in the following, explicit mention of the background theory is usually suppressed.

Example 4.2.3. Without spelling out the background theory, one would expect that the standard predicates $(\wedge_2\ p\ q)$ and $(\wedge_2\ q\ p)$ are equivalent, where p and q are standard predicates. Similarly, $(setExists_2\ p\ q)$ and $(setExists_2\ q\ p)$ are equivalent. Less obviously, $(domCard\ p) \circ (> 0)$ and $(setExists_1\ p)$ are equivalent.

It is important to remove as much of this redundancy in standard predicates as possible. Ideally, one would like to be able to determine just *one* representative from each class of equivalent predicates. However, determining the equivalent predicates is undecidable, so one usually settles for some easily checked syntactic conditions that reveal equivalence of predicates. Thus these syntactic conditions are sufficient, but not necessary, for equivalence. These considerations motivate the next definition.

Definition 4.2.7. A transformation f is *symmetric* if it has a signature of the form

$$f : (\varrho \to \Omega) \to \cdots \to (\varrho \to \Omega) \to \mu \to \sigma,$$

and, whenever $f\ p_1 \ldots p_k$ is a standard predicate, it follows that $f\ p_1 \ldots p_k$ and $f\ p_{i_1} \ldots p_{i_k}$ are equivalent, for all permutations i of $\{1, \ldots, k\}$, where k is the rank of f.

Note that if $f\ p_1 \dots p_k$ is a standard predicate, then so is $f\ p_{i_1} \dots p_{i_k}$. It is common for a symmetric transformation to have Ω as its target; however, this does not have to be the case. Clearly, every transformation of rank k, where $k \leq 1$, is (trivially) symmetric. Furthermore, the transformations $setExists_n$ and $\wedge_n$ are symmetric. However, *isRectangle* is not symmetric since it distinguishes the length argument from the breadth argument.

Since any permutation of the predicate arguments of a symmetric transformation produces an equivalent function, it is advisable to choose one particular order of arguments and ignore the others. For this purpose, a total order on standard predicates is defined and then arguments for symmetric transformations are chosen in increasing order according to this total order. To define the total order on standard predicates, one must start with a total order on transformations. Therefore, it is supposed that the transformations are ordered according to some (arbitrary) strict total order $<$.

In preparation for the definition of $\prec$, a structural result about standard predicates is needed.

Proposition 4.2.5. *Let $p, q \in \mathbf{S}$, where p is $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \dots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$ and q is $(g_1\ q_{1,1} \dots q_{1,s_1}) \circ \dots \circ (g_r\ q_{r,1} \dots q_{r,s_r})$. Suppose that p and q are (syntactically) distinct. Then exactly one of the following alternatives holds.*

1. *One of p or q is a proper prefix of the other.*
2. *There exists i such that the transformation f_i in p and the transformation g_i in q are distinct, and p and q agree to the left of f_i and g_i.*
3. *There exist i and j_i such that p_{i,j_i} in p and q_{i,j_i} in q are distinct, and p and q agree to the left of p_{i,j_i} and q_{i,j_i}.*

Proof. The proof is by induction on the maximum of the number of (occurrences of) transformations in p and the number in q.

Suppose first that f_1 is distinct from g_1. Then the second alternative holds. Otherwise, f_1 and g_1 are identical, which gives rise to two cases: either the arguments to f_1 in p and g_1 in q are pairwise identical or they are not. In the first case, consider the suffixes of p and q obtained by removing the common prefix $(f_1\ p_{1,1} \dots p_{1,k_1})$. If one of the suffixes is empty, then the first alternative holds; otherwise, one can apply the inductive hypothesis to the suffixes to obtain the result. In the second case, the leftmost pair of p_{1,j_1} and q_{1,j_1} that are distinct gives the third alternative. □

Example 4.2.4. As an illustration of the first alternative in the preceding proposition, consider p and $p \circ \neg$, for some standard predicate p. For the third alternative, consider $(\wedge_2\ p\ q)$ and $(\wedge_2\ (p \circ \neg)\ q)$, for some standard predicates p and q.

The following definition of the relation $p \prec q$ uses induction on the maximum of the number of (occurrences of) transformations in p and the number in q. To emphasise: the definition of $\prec$ depends upon $<$.

Definition 4.2.8. The binary relation $\prec$ on $\mathbf{S}$ is defined inductively as follows. Let $p, q \in \mathbf{S}$, where p is $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \dots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$ and q is $(g_1\ q_{1,1} \dots q_{1,s_1}) \circ \dots \circ (g_r\ q_{r,1} \dots q_{r,s_r})$. Then $p \prec q$ if one of the following holds.

1. p is a proper prefix of q.
2. There exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i.
3. There exist i and j_i such that $p_{i,j_i} \prec q_{i,j_i}$, and p and q agree to the left of p_{i,j_i} and q_{i,j_i}.

Proposition 4.2.6. *The relation $\prec$ is a strict total order on* $\mathbf{S}$.

Proof. Let $p, q, r \in \mathbf{S}$, where p is $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \dots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$, q is $(g_1\ q_{1,1} \dots q_{1,s_1}) \circ \dots \circ (g_u\ q_{u,1} \dots q_{u,s_u})$ and r is $(h_1\ r_{1,1} \dots r_{1,t_1}) \circ \dots \circ (h_v\ r_{v,1} \dots r_{v,t_v})$. First, I show by induction on the number of (occurrences of) transformations in p that $p \not\prec p$. For this, note that p cannot be a proper prefix of itself and there cannot exist i such that $f_i < f_i$, since $<$ is a strict total order. Also there cannot exist i and j_i such that $p_{i,j_i} \prec p_{i,j_i}$, by the induction hypothesis.

Next I show by induction on the maximum of the number of transformations in p and the number in q that $p \prec q$ implies $q \not\prec p$. Thus suppose that $p \prec q$. If p is a proper prefix of q, then it is clear that $q \not\prec p$. If there exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i, then it is clear that $q \not\prec p$, since $<$ is a strict total order. Finally, if there exist i and j_i such that $p_{i,j_i} \prec q_{i,j_i}$, and p and q agree to the left of p_{i,j_i} and q_{i,j_i}, then $q \not\prec p$ since, by the induction hypothesis, it follows that $q_{i,j_i} \not\prec p_{i,j_i}$.

Now I show by induction on the maximum of the number of transformations in p, the number in q, and the number in r that $p \prec q$ and $q \prec r$ imply that $p \prec r$. There are three cases to consider, corresponding to the three cases in the definition of $p \prec q$.

Suppose that p is a proper prefix of q. If q is a proper prefix of r, then p is a proper prefix of r and so $p \prec r$. If there exists i such that $g_i < h_i$, and q and r agree to the left of g_i and h_i, then either p is a proper prefix of r or $f_i < h_i$ and p and r agree to the left of f_i and h_i, and so $p \prec r$. If there exist i and j_i such that $q_{i,j_i} \prec r_{i,j_i}$, and q and r agree to the left of q_{i,j_i} and r_{i,j_i}, then either p is a proper prefix of r or $p_{i,j_i} \prec r_{i,j_i}$ and p and r agree to the left of p_{i,j_i} and r_{i,j_i}, and so $p \prec r$.

For the second case, suppose there exists i such that $f_i < g_i$, and p and q agree to the left of f_i and g_i, If q is a proper prefix of r, then $f_i < h_i$, and p and r agree to the left of f_i and h_i, and so $p \prec r$. If there exists i' such that $g_{i'} < h_{i'}$, and q and r agree to the left of $g_{i'}$ and $h_{i'}$, then, for $i'' = min(i, i')$, $f_{i''} < h_{i''}$ and p and r agree to the left of $f_{i''}$ and $h_{i''}$, and so $p \prec r$. Finally, suppose that there exist i' and $j_{i'}$ such that $q_{i',j_{i'}} \prec r_{i',j_{i'}}$, and q and r agree to the left of $q_{i',j_{i'}}$ and $r_{i',j_{i'}}$. If $i \leq i'$, then $f_i < h_i$ and p and r agree to the

left of f_i and h_i. If $i' < i$, then $p_{i',j_{i'}} \prec r_{i',j_{i'}}$ and p and r agree to the left of $p_{i',j_{i'}}$ and $r_{i',j_{i'}}$. In either case, $p \prec r$.

For the third case, suppose there exist i and j_i such that $p_{i,j_i} \prec q_{i,j_i}$, and p and q agree to the left of p_{i,j_i} and q_{i,j_i}. If q is a proper prefix of r, then $p_{i,j_i} \prec r_{i,j_i}$, and p and r agree to the left of p_{i,j_i} and r_{i,j_i}, and so $p \prec r$. Suppose that there exists i' such that $g_{i'} < h_{i'}$, and q and r agree to the left of $g_{i'}$ and $h_{i'}$. If $i' \leq i$, then $f_{i'} < h_{i'}$ and p and r agree to the left of $f_{i'}$ and $h_{i'}$. If $i < i'$, then $p_{i,j_i} \prec r_{i,j_i}$ and p and r agree to the left of p_{i,j_i} and r_{i,j_i}. In either case, $p \prec r$. Finally, suppose there exist i' and $j_{i'}$ such that $q_{i',j_{i'}} \prec r_{i',j_{i'}}$, and q and r agree to the left of $q_{i',j_{i'}}$ and $r_{i',j_{i'}}$. Put $i'' = min(i, i')$ and let $j_{i''}$ be j_i if $i < i'$, or $j_{i'}$ if $i' < i$, or $min(j_{i'}, j_{i''})$, otherwise. Then $p_{i'',j_{i''}} \prec r_{i'',j_{i''}}$ and p and r agree to the left of $p_{i'',j_{i''}}$ and $r_{i'',j_{i''}}$, and so $p \prec r$. (Here the induction hypothesis is used if $i = i'$ and $j_i = j_{i'}$.)

Thus $\prec$ is a strict partial order.

Finally, I show that $\prec$ is total, that is, for any standard predicates p and q, either $p = q$ or $p \prec q$ or $q \prec p$. The proof proceeds by induction on the maximum of the number of predicates in p and the number in q. Suppose that p and q are distinct. By Proposition 4.2.5, one of p or q is a proper prefix of the other; or there exists i such that the transformation f_i in p and the transformation g_i in q are distinct, and p and q agree to the left of f_i and g_i; or there exist i and j_i such that p_{i,j_i} in p and q_{i,j_i} in q are distinct, and p and q agree to the left of p_{i,j_i} and q_{i,j_i}. In the first case, either $p \prec q$ or $q \prec p$. In the second case, since $<$ is a total order, either $f_i < g_i$ or $g_i < f_i$, and hence either $p \prec q$ or $q \prec p$. In the third case, by the induction hypothesis, either $p_{i,j_i} \prec q_{i,j_i}$ or $q_{i,j_i} \prec p_{i,j_i}$, and so once again either $p \prec q$ or $q \prec p$. □

The relation $\preceq$ is defined by $p \preceq q$ if either $p = q$ or $p \prec q$. Clearly, $\preceq$ is a total order on **S**.

4.3 Regular Predicates

Now the class of regular predicates can be defined by induction on the number of transformations in a predicate.

Definition 4.3.1. A standard predicate $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$ is *regular* if p_{i,j_i} is a regular predicate, for $i = 1, \dots, n$ and $j_i = 1, \dots, k_i$, and f_i is symmetric implies that $p_{i,1} \preceq \cdots \preceq p_{i,k_i}$, for $i = 1, \dots, n$.

The set of all regular predicates is denoted by **R**.

Example 4.3.1. Going back to Example 4.1.15,

$$vertices \circ (domCard\ (vertex \circ (\wedge_2\ p\ q))) \circ (> 7)$$

is a regular predicate iff $p \preceq q$, and p and q are regular predicates.

The next result shows that each standard predicate is equivalent to a regular predicate and hence attention can be confined to the generally much smaller class of regular predicates.

Proposition 4.3.1. *For every* $p \in \mathbf{S}$, *there exists* $q \in \mathbf{R}$ *such that* p *and* q *are equivalent.*

Proof. The existence of the regular predicate q is shown by induction on the number of transformations in p. Let p be $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \dots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$. By the induction hypothesis, for each $i = 1, \dots, n$ and $j_i = 1, \dots, k_i$, there is a regular predicate p'_{i,j_i} such that p'_{i,j_i} and p_{i,j_i} are equivalent. Let r be $(f_1\ p'_{1,1} \dots p'_{1,k_1}) \circ \dots \circ (f_n\ p'_{n,1} \dots p'_{n,k_n})$. Then p and r are equivalent. Now, since $\preceq$ is a total order on standard predicates of the same type,, for each symmetric transformation f_i ($i \in \{1, \dots, n\}$) in r, one can sort the arguments $p'_{i,1}, \dots, p'_{i,k_i}$ according to $\preceq$ to obtain the desired regular predicate q such that p and q are equivalent. □

One cannot expect the regular predicate q in Proposition 4.3.1 to be (syntactically) unique. To see this, simply note that $(domCard\ top) \circ (> 0)$ and $(setExists_1\ top)$ are regular predicates, and are equivalent.

On the other hand, using the proof of Proposition 4.3.1 as a basis, one can give an algorithm for constructing a regular predicate equivalent to some given standard predicate. This algorithm is given in Fig. 4.2 below.

```
function Regularise(p) returns a regular predicate q such that p and q are
                                    equivalent;
input: p, a standard predicate (f_1 p_{1,1} ... p_{1,k_1}) ∘ ··· ∘ (f_n p_{n,1} ... p_{n,k_n});

for i = 1,...,n do
    for j_i = 1,...,k_i do
        p̄_{i,j_i} := Regularise(p_{i,j_i});
    if f_i is symmetric
    then
        [q_{i,1},...,q_{i,k_i}] := Sort([p̄_{i,1},...,p̄_{i,k_i}]);
        p_i := (f_i q_{i,1} ... q_{i,k_i});
    else
        p_i := (f_i p̄_{i,1} ... p̄_{i,k_i});
return p_1 ∘ ··· ∘ p_n;
```

Fig. 4.2. Algorithm for regularising a standard predicate

Definition 4.3.2. The standard predicate q given by the algorithm in Fig. 4.2 is called the *regularisation* of the standard predicate p. The regularisation of p is denoted by $\overline{p}$.

Proposition 4.3.2.

1. *The regularisation of a standard predicate is unique.*
2. *The regularisation of a standard predicate has the same type as the standard predicate.*
3. *The regularisation of a standard predicate is a regular predicate.*
4. *A standard predicate and its regularisation are equivalent.*
5. *The regularisation of a regular predicate is itself.*
6. *If two standard predicates have the same regularisation, then they contain the same number of transformations.*

Proof. Each of these parts is a straightforward induction argument on the number of transformations in the standard predicate. □

A more detailed look at the properties of regularisation is now undertaken. The first step is to give another algorithm for finding the regularisation of a standard predicate. This algorithm is given in Fig. 4.3 below. The basic idea of the algorithm is simple: whenever two adjacent regular predicates are 'not in order', transpose them, until no such transpositions are possible.

function *Regularise2*(p) **returns** a regular predicate q such that p and q are equivalent;
input: p, a standard predicate;

$q := p$;
while there exist a standard predicate that is a subterm in q of the form $(g\ q_1 \dots q_m)$ and $i \in \{1, \dots, m-1\}$ such that g is symmetric, q_i and q_{i+1} are regular, and $q_{i+1} \prec q_i$ **do**
    $q := q[(g\ q_1 \dots q_m)/(g\ q_1 \dots q_{i-1}\ q_{i+1}\ q_i\ q_{i+2} \dots q_m)]$;
return q;

Fig. 4.3. Another algorithm for regularising a standard predicate

Proposition 4.3.3. *The algorithm of Fig. 4.3 terminates and returns the regularisation of the input standard predicate.*

Proof. First, termination is demonstrated. The proof is by induction on the number of transformations in the input standard predicate p. Thus suppose the algorithm terminates for standard predicates that have $< m$ transformations and p has m transformations. Let p have the form $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$. By the induction hypothesis, the algorithm terminates on p_{i,j_i}, for each i and j_i. Thus it is only necessary to show that the algorithm terminates when putting the regularisations of $p_{i,1}, \dots, p_{i,k_i}$ in order according to $\prec$, for $i = 1, \dots, n$. But this follows since each transposition puts two

such regular predicates in the correct order with respect to one another and later transpositions do not change this.

The proof that the algorithm returns the regularisation of the input standard predicate p is also by induction on the number of transformations in p. Thus suppose the result holds for standard predicates that have $< m$ transformations and p has m transformations. By the induction hypothesis, the algorithm returns the regularisation of p_{i,j_i}, for all i and j_i. Then, for symmetric f_i, it is clear that by doing the transpositions the algorithm puts $p_{i,1}, \ldots p_{i,k_i}$ in the correct order according to $\preceq$, for $i = 1, \ldots, n$. □

Example 4.3.2. Let f, g, h, i, j, and k be transformations such that f, h, and k are symmetric and $(f\ (h\ j\ i)\ g) \circ (k\ i\ g)$ is a standard predicate. Suppose the order on the transformations is $f < g < h < i < j < k$. Then the algorithm of Fig. 4.3 could obtain the regularisation of this standard predicate by a sequence of transpositions that produces the following sequence of standard predicates.

$(f\ (h\ j\ i)\ g) \circ (k\ i\ g)$
$(f\ (h\ i\ j)\ g) \circ (k\ i\ g)$
$(f\ g\ (h\ i\ j)) \circ (k\ i\ g)$
$(f\ g\ (h\ i\ j)) \circ (k\ g\ i)$.

Thus $(f\ g\ (h\ i\ j)) \circ (k\ g\ i)$ is the regularisation of $(f\ (h\ j\ i)\ g) \circ (k\ i\ g)$. Of course, a number of other sequences of transpositions also give the regularisation.

The next issue is to track the 'movement' of a subterm during regularisation of the input standard predicate. Consider a single step of the regularisation algorithm of Fig. 4.3. Suppose q is a subterm of the current standard predicate being considered by the algorithm that is itself a standard predicate and has occurrence o. When one step of the algorithm is applied, several things can happen to q and o. If the step concerns transposing two subterms that are disjoint from q, then q is unchanged in the next standard predicate and has the same occurrence. If the step concerns transposing two subterms that are subterms of q, then q is modified but its occurrence is unchanged. Finally, if the step concerns transposing two subterms one of which has q as a subterm, then q is unchanged but its occurrence changes. There are no other cases. Consequently, there is a sequence of pairs of subterms and their occurrences of the form $(s_1, o_1), \ldots, (s_m, o_m)$, where s_1 is the initial subterm q and s_m is a subterm in the regularisation, and each (s_i, o_i) is obtained from (s_{i-1}, o_{i-1}) by a single step of the algorithm.

Definition 4.3.3. The subterm s_m is called the *transpose* of s_1 and the occurrence o_m is called the *transpose* of o_1.

If q is a subterm at occurrence o, then its transpose is denoted by q^t and the occurrence of the transpose is denoted by o^t.

Proposition 4.3.4. *Let p and q be standard predicates such that q is a subterm of p at occurrence o. Then the following hold.*

1. *The transpose of q is the regularisation of q.*
2. *The transpose of o is unique.*
3. *The positional type of q^t in $\overline{p}$ is the same as the positional type of q in p.*

Proof. 1. Since the regularisation of p is regular, it follows that q^t being a subterm that is a standard predicate in this regular predicate is itself regular. Thus q^t is the output of the algorithm of Fig. 4.3 applied to q and hence is the regularisation of q.

2. The proof is by induction on the number of transformations in p. Suppose the result holds for standard predicates that contain $< m$ transformations and p contains m transformations. Let p be $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$. If q is a suffix of p, then it is clear that $o^t = o$. Similarly, if q is $(f_i\ p_{i,1} \ldots p_{i,k_i})$, for some i, then it is clear that $o^t = o$. Otherwise, q is a subterm of some p_{i,j_i}. By the induction hypothesis, the transpose of the occurrence of q in p_{i,j_i} is unique. Since the transpose of the occurrence of p_{i,j_i} in p is uniquely determined by $\prec$, the result follows.

3. This is a straightforward induction argument. □

Proposition 4.3.5. *Let p and p' be standard predicates such that $\overline{p} = \overline{p'}$. Suppose that q is a standard predicate which is a subterm of p at occurrence o and q' is a standard predicate which is a subterm of p' at occurrence o' such that $o^t = o'^t$. Let r and r' be standard predicates such that $\overline{r} = \overline{r'}$ and $p[q/r]$ and $p'[q'/r']$ are standard predicates. Then $\overline{p[q/r]} = \overline{p'[q'/r']}$.*

Proof. Proposition 4.3.2 shows that p and p' have the same number of transformations. The proof is by induction on the number of transformations in p and p'. Suppose that the result holds for pairs of standard predicates that contain $< m$ transformations and that p and p' contain m transformations. Let p be

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n}).$$

Then, since $\overline{p} = \overline{p'}$, p' must have the form

$$(f_1\ p'_{1,1} \ldots p'_{1,k_1}) \circ \cdots \circ (f_n\ p'_{n,1} \ldots p'_{n,k_n}).$$

If q is a suffix $(f_i\ p_{i,1} \ldots p_{i,k_i}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$, then q' must be the suffix $(f_i\ p'_{i,1} \ldots p'_{i,k_i}) \circ \cdots \circ (f_n\ p'_{n,1} \ldots p'_{n,k_n})$, since $o^t = o'^t$, and the result follows. Similarly, if q is $(f_i\ p_{i,1} \ldots p_{i,k_i})$, then q' must be $(f_i\ p'_{i,1} \ldots p'_{i,k_i})$, since $o^t = o'^t$, and the result follows. Finally, suppose that q is a subterm of some p_{i,j_i}. Then, since $o^t = o'^t$, q' must be a subterm of some p'_{i,l_i}. Since $\overline{p_{i,j_i}} = \overline{p'_{i,l_i}}$, by the induction hypothesis, $\overline{p_{i,j_i}[q/r]} = \overline{p'_{i,l_i}[q'/r']}$. Thus $\overline{p[q/r]} = \overline{p'[q'/r']}$. □

Before moving on to other issues, note that there is scope for employing syntactic conditions other than symmetry to determine equivalence of predicates. Furthermore, notwithstanding the undecidability of the problem, one could even employ a theorem prover to attempt to establish equivalence. If the cost of proving a theorem to show that two predicates are equivalent is less than the cost of redundantly using both predicates in some application, then the theorem-proving approach is worthwhile.

The final issue is that of the finiteness of the class of standard predicates. If there are infinitely many transformations, then clearly there can be infinitely many standard predicates. For example, if all predicates on integers of the form $(> n)$, for $n = 1, 2, 3, \ldots$, are included in the class of transformations, then (trivially) there are infinitely many standard (and also regular) predicates. Even in the case when there are only finitely many transformations, there may still be infinitely many standard predicates. To see this, consider the transformation $leftTree : BTree\ \mu \to BTree\ \mu$. Here $(BTree\ \mu)$ is the type of binary trees whose nodes have type μ and the meaning of *leftTree* is that it returns the left subtree of its argument. Then compositions of *leftTree* with itself of unbounded length can be formed, and so there are infinitely many standard predicates. The next proposition gives obvious conditions under which the class of standard predicates is finite.

Proposition 4.3.6. *Suppose that the class of transformations is finite and the number of times each transformation may appear in a standard predicate is bounded. Then the number of standard, and hence regular, predicates is finite.*

Proof. The product of the number of transformations and the maximum number of times any transformation can appear gives an upper bound on the number of transformations that can appear in any standard predicate. Hence the number of standard predicates is finite. □

4.4 Predicate Rewrite Systems

This section addresses the central issue of the construction of standard predicates using predicate rewrite systems.

Definition 4.4.1. A *predicate rewrite system* is a finite relation $\rightarrowtail$ on $\mathbf{S}$ satisfying the following two properties.

1. For each $p \rightarrowtail q$, the type of p is more general than the type of q.
2. For each $p \rightarrowtail q$, there does not exist $s \rightarrowtail t$ such that q is a proper subterm of s.

If $p \rightarrowtail q$, then $p \rightarrowtail q$ is called a *predicate rewrite*, p the *head*, and q the *body* of the predicate rewrite.

The second condition of Definition 4.4.1 states that no body of a rewrite is a proper subterm of the head of any rewrite. In practice, the heads of rewrites are usually standard predicates consisting of just one transformation, such as *top*. In this case, the second condition is automatically satisfied, since the heads have no proper subterms at all.

Proposition 4.4.1. *Let p, q, and r be standard predicates, and q an eligible subterm of p. Then $p[q/r]$ is a standard predicate iff the type of r and the positional type of q in p are unifiable.*

Proof. By Proposition 4.2.4, $p[q/r]$ is a standard predicate iff $p[q/r]$ is a term. Then, by Proposition 2.5.7, $p[q/r]$ is a term iff the type of r and the positional type of q in p are unifiable. □

Definition 4.4.2. Let $\rightarrowtail$ be a predicate rewrite system and p a standard predicate. An eligible subterm r of p is a *redex* with respect to $\rightarrowtail$ if there exists a predicate rewrite $r \rightarrowtail b$ such that the type of b and the positional type of r in p are unifiable. In this case, r is said to be a redex *via* $r \rightarrowtail b$.

It follows from Proposition 4.4.1 that r is a redex iff $p[r/b]$ is a standard predicate. The phrase 'redex with respect to $\rightarrowtail$' is usually abbreviated to simply 'redex' with the predicate rewrite system understood.

Definition 4.4.3. Let $\rightarrowtail$ be a predicate rewrite system, and p and q standard predicates. Then q is obtained by a *predicate derivation step* from p using $\rightarrowtail$ if there is a redex r via $r \rightarrowtail b$ in p and $q = p[r/b]$. The redex r is called the *selected* redex.

Proposition 4.4.2. *Let $\rightarrowtail$ be a predicate rewrite system, and p and q be standard predicates such that q is obtained by a predicate derivation step from p. Then the type of p is more general than the type of q.*

Proof. Suppose that q is obtained by a predicate derivation step from p via $r \rightarrowtail b$. Thus q is $p[r/b]$ and the type of r is more general than the type of b. Consequently, by Proposition 2.5.8, the type of p is more general than the type of $p[r/b]$. □

Proposition 4.4.3. *Let $\rightarrowtail$ be a predicate rewrite system, and p and q be standard predicates such that q is obtained by a predicate derivation step from p via $r \rightarrowtail b$, where r is a subterm of p at occurrence o. Then the following hold.*

1. *No occurrence of a redex in q is a proper prefix of o.*
2. *For each redex s in q at occurrence o', say, either s is a redex in p at occurrence o' such that o and o' are disjoint or s is a redex in b at occurrence o'', where $o' = oo''$.*

Intuitively, all redexes (more precisely, the underlying standard predicates) in q either are already present in p or are introduced through b.

Proof. 1. If there was an occurrence of a redex in q that was a proper prefix of the occurrence of the redex r in p, then the body b of the rewrite $r \rightarrowtail b$ would be a proper subterm of the head of some rewrite, contradicting the second condition in Definition 4.4.1.

2. According to Part 1, if s is a redex at occurrence o' in q, then o' cannot be a proper prefix of o. Thus either there exists o'' such that $o' = oo''$ or neither o nor o' is a prefix of the other.

In the first case, s is a subterm of b at occurrence o''. It follows from Proposition 2.5.2 that the positional type of s in b is weaker than the positional type of s in q. Thus s is a redex in b.

In the second case, since the type of r is more general than the type of b by the first condition in Definition 4.4.1, using Proposition 2.5.2, one can show that the positional type of s in p is weaker than the positional type of s in q. Thus s is a redex in p. □

If the second condition of Definition 4.4.1 is dropped, then Proposition 4.4.3 no longer holds.

Example 4.4.1. Consider a transformation $f : (a \rightarrow \Omega) \rightarrow a \rightarrow \Omega$, where a is a parameter. Let $\rightarrowtail$ be the relation

$$\begin{aligned} &top \rightarrowtail bottom \\ &f\ bottom \rightarrowtail bottom. \end{aligned}$$

Then $f\ bottom$ is obtained from $f\ top$ by a predicate rewrite step using the rewrite $top \rightarrowtail bottom$. However, $f\ bottom$ is a redex in $f\ bottom$ and thus Part 1 of Proposition 4.4.3 does not hold. For $\rightarrowtail$, the body of both rewrites is a proper subterm of the head of the second rewrite. Thus $\rightarrowtail$ satisfies the first, but not the second, condition of Definition 4.4.1.

Definition 4.4.4. A *predicate derivation* with respect to a predicate rewrite system $\rightarrowtail$ is a finite sequence $\langle p_0, p_1, \ldots, p_n \rangle$ of standard predicates such that p_i is obtained by a derivation step from p_{i-1} using $\rightarrowtail$, for $i = 1, \ldots, n$. The *length* of the predicate derivation is n. The standard predicate p_0 is called the *initial predicate* and the standard predicate p_n is called the *final predicate.*

Usually the initial predicate is top, the weakest predicate.

Example 4.4.2. Consider the keys example of Sect. 1.3 again. Let the transformations be as follows. (The transformation $top : a \rightarrow \Omega$ is always taken for granted.)

$$(= Abloy) : Make \to \Omega$$
$$(= Chubb) : Make \to \Omega$$
$$(= Rubo) : Make \to \Omega$$
$$(= Yale) : Make \to \Omega$$
$$(= 2) : NumProngs \to \Omega$$
$$(= 3) : NumProngs \to \Omega$$
$$(= 4) : NumProngs \to \Omega$$
$$(= 5) : NumProngs \to \Omega$$
$$(= 6) : NumProngs \to \Omega$$
$$(= Short) : Length \to \Omega$$
$$(= Medium) : Length \to \Omega$$
$$(= Long) : Length \to \Omega$$
$$(= Narrow) : Width \to \Omega$$
$$(= Normal) : Width \to \Omega$$
$$(= Broad) : Width \to \Omega$$
$$projMake : Key \to Make$$
$$projNumProngs : Key \to NumProngs$$
$$projLength : Key \to Length$$
$$projWidth : Key \to Width$$
$$setExists_1 : (Key \to \Omega) \to Bunch \to \Omega$$
$$\wedge_4 : (Key \to \Omega) \to (Key \to \Omega) \to (Key \to \Omega) \to (Key \to \Omega) \to Key \to \Omega.$$

Let the predicate rewrite system be as follows.

$$top \rightarrowtail setExists_1\ (\wedge_4\ (projMake \circ top)\ (projNumProngs \circ top)$$
$$(projLength \circ top)\ (projWidth \circ top))$$
$$top \rightarrowtail (= Abloy)$$
$$top \rightarrowtail (= Chubb)$$
$$top \rightarrowtail (= Rubo)$$
$$top \rightarrowtail (= Yale)$$
$$top \rightarrowtail (= 2)$$
$$top \rightarrowtail (= 3)$$
$$top \rightarrowtail (= 4)$$
$$top \rightarrowtail (= 5)$$
$$top \rightarrowtail (= 6)$$
$$top \rightarrowtail (= Short)$$
$$top \rightarrowtail (= Medium)$$

$$top \rightarrowtail (= Long)$$
$$top \rightarrowtail (= Narrow)$$
$$top \rightarrowtail (= Normal)$$
$$top \rightarrowtail (= Broad).$$

It is clear that the above set of rewrites satisfies the two conditions in Definition 4.4.1. Then the following is a predicate derivation with respect to this predicate rewrite system.

$$\begin{aligned}
&top\\
&setExists_1\ (\wedge_4\ (projMake \circ top)\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ top)\ (projWidth \circ top))\\
&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ top)\ (projWidth \circ top))\\
&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ (= Medium))\ (projWidth \circ top))\\
&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad)))\\
&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ (= 6))\\
&\qquad\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad))).
\end{aligned}$$

The first predicate rewrite is used in the first step. (It is intended that the derivation provide a predicate of type $Bunch \rightarrow \Omega$.) For the second step, the redex *top* has positional type $Make \rightarrow \Omega$ which is the same as the type of the body of the predicate rewrite $top \rightarrowtail (= Abloy)$. Thus this occurrence of *top* is indeed a redex via this predicate rewrite. The remaining steps are similar.

Proposition 4.4.4. *Let* $\rightarrowtail$ *be a predicate rewrite system and* $\langle p_0, p_1, \ldots, p_n \rangle$ *a predicate derivation using* $\rightarrowtail$. *Then the type of* p_0 *is more general than the type of* p_n.

Proof. The proof is an easy induction argument on the length of the predicate derivation using Proposition 4.4.2. □

In some later results, the concept of a predicate subderivation will be needed.

Definition 4.4.5. Let $\rightarrowtail$ be a predicate rewrite system and $\langle p_0, p_1, \ldots, p_n \rangle$ a predicate derivation with respect to $\rightarrowtail$. Then the *predicate subderivation* $\langle q_0, q_1, \ldots, q_m \rangle$ with *initial predicate* q_0, where q_j is a standard predicate that is a subterm of p_{i_j}, for $j = 0, \ldots, m$, and $i_0 < i_1 < \cdots < i_m$, is defined by induction on n as follows.

If $n = 0$, then q_0 must be a subterm of p_0 and the predicate subderivation with initial predicate q_0 is $\langle q_0 \rangle$.

Suppose now that predicate subderivations for predicate derivations up to length n have been defined. Let $\langle p_0, p_1, \ldots, p_n, p_{n+1}\rangle$ be a predicate derivation of length $n+1$ and q_0 be a standard predicate that is a subterm of p_{i_0}, for some $i_0 \in \{0, \ldots, n+1\}$. There are two cases to consider.

1. If $i_0 \leq n$, then the predicate subderivation $\langle q_0, q_1, \ldots, q_m\rangle$ with initial predicate q_0 for the predicate derivation $\langle p_0, p_1, \ldots, p_n\rangle$ is defined. If the redex r selected in p_n is a subterm of q_m, then the predicate subderivation with initial predicate q_0 for the predicate derivation $\langle p_0, p_1, \ldots, p_n, p_{n+1}\rangle$ is $\langle q_0, q_1, \ldots, q_m, q_m[r/b]\rangle$, where $r \rightarrowtail b$ is the rewrite used. Otherwise, the predicate subderivation with initial predicate q_0 for the predicate derivation $\langle p_0, p_1, \ldots, p_n, p_{n+1}\rangle$ is $\langle q_0, q_1, \ldots, q_m\rangle$.
2. If $i_0 = n+1$, then predicate subderivation with initial predicate q_0 for the predicate derivation $\langle p_0, p_1, \ldots, p_n, p_{n+1}\rangle$ is $\langle q_0\rangle$.

Example 4.4.3. For the predicate derivation in Example 4.4.2, the following is a predicate subderivation.

$$
\begin{aligned}
&\wedge_4\ (\mathit{projMake} \circ (= \mathit{Abloy}))\ (\mathit{projNumProngs} \circ \mathit{top}) \\
&\qquad\qquad\qquad\qquad (\mathit{projLength} \circ \mathit{top})\ (\mathit{projWidth} \circ \mathit{top}) \\
&\wedge_4\ (\mathit{projMake} \circ (= \mathit{Abloy}))\ (\mathit{projNumProngs} \circ \mathit{top}) \\
&\qquad\qquad\qquad (\mathit{projLength} \circ (= \mathit{Medium}))\ (\mathit{projWidth} \circ \mathit{top}) \\
&\wedge_4\ (\mathit{projMake} \circ (= \mathit{Abloy}))\ (\mathit{projNumProngs} \circ \mathit{top}) \\
&\qquad\qquad (\mathit{projLength} \circ (= \mathit{Medium}))\ (\mathit{projWidth} \circ (= \mathit{Broad})) \\
&\wedge_4\ (\mathit{projMake} \circ (= \mathit{Abloy}))\ (\mathit{projNumProngs} \circ (= 6)) \\
&\qquad\qquad (\mathit{projLength} \circ (= \mathit{Medium}))\ (\mathit{projWidth} \circ (= \mathit{Broad})).
\end{aligned}
$$

Proposition 4.4.5. *Let $\rightarrowtail$ be a predicate rewrite system and $\langle p_0, p_1, \ldots, p_n\rangle$ a predicate derivation with respect to $\rightarrowtail$. Suppose $\langle q_0, q_1, \ldots, q_m\rangle$ is a predicate subderivation with initial predicate q_0 for $\langle p_0, p_1, \ldots, p_n\rangle$. Then $\langle q_0, q_1, \ldots, q_m\rangle$ is a predicate derivation with respect to $\rightarrowtail$.*

Proof. The proof is by induction on the length n of the predicate derivation. If $n = 0$, the result is obvious. Suppose now that the result holds for predicate derivations of length up to n. Let $\langle p_0, p_1, \ldots, p_{n+1}\rangle$ be a predicate derivation of length $n+1$. Let q_0 be a standard predicate that is a subterm of p_{i_0}. If $i_0 = n+1$, the result is obvious. Otherwise, $i_0 \leq n$. By the induction hypothesis, the predicate subderivation $\langle q_0, q_1, \ldots, q_m\rangle$ with initial predicate q_0 is a predicate derivation. If the redex selected in p_n is not a subterm of q_m, the result follows. Suppose finally that the selected redex r in p_n is a subterm of q_m. Then r is a redex in q_m, since, by Proposition 2.5.2, the positional type of r in q_m is more general than the positional type of r in p_n. Thus, if the rewrite used is $r \rightarrowtail b$, then $q_m[r/b]$ is a standard predicate. Putting $q_{m+1} = q_m[r/b]$, it follows that $\langle q_0, q_1, \ldots, q_m, q_{m+1}\rangle$ is a predicate derivation, as required. □

In machine learning applications, a predicate rewrite system is used to generate a search space of standard predicates. Starting from some predicate p_0, one generates all the standard predicates that can be obtained by a predicate derivation step from p_0, then all the standard predicates that can be obtained from those by a predicate derivation step, and so on. A path in the search space is a sequence of standard predicates each of which (except for the first) can be obtained from its predecessor by a predicate derivation step. Thus each path in the search space is a predicate derivation. As shall be shown later, the search space may not be a tree since there may be more than one path to a particular predicate. An algorithm for generating the search space is given in Fig. 4.4.

function *Enumerate*$(\rightarrowtail, p_0)$ **returns** the set of standard predicates in the search space rooted at p_0 using $\rightarrowtail$;

input: $\rightarrowtail$, a predicate rewrite system;
p_0, a standard predicate;

predicates := {};
openList := $[p_0]$;
while *openList* $\neq$ [] **do**
  p := *head*(*openList*);
  openList := *tail*(*openList*);
  predicates := *predicates* $\cup$ $\{p\}$;
  for each redex r via $r \rightarrowtail b$, for some b, in p **do**
    $q := p[r/b]$;
    openList := *openList* ++ $[q]$;
return *predicates*;

Fig. 4.4. Algorithm for enumerating standard predicates

Typically, p_0 is *top*. Since the intention is usually to generate only predicates that are applicable to a particular closed type (the type of the individuals in the application), there can be some ambiguity, especially at the top level, about which rewrites should be used. Thus there may be an implied restriction on the choice of rewrites that are used in derivation steps to enforce this property. For example, in Example 4.4.2, if one is only interested in predicates on bunches of keys, then only the first rewrite would be used at the root *top*. After that, the types of the redexes are sufficiently constrained by their position in the standard predicates that are generated and there is no more ambiguity.

4.5 The Implication Preorder

The next major issue is that of determining whether one predicate logically implies (that is, is stronger than) another. This relationship between predicates plays a crucial role in structuring the search space of predicates in applications. The first definition needed for this is that of the implication preorder for standard predicates. The binary relation $\Leftarrow$ on $\mathbf{S}$ is defined as follows.

Definition 4.5.1. Let $\mathcal{B}$ be the background theory, and $p, q \in \mathbf{S}$. Then $p \Leftarrow q$ if the type of p is more general than the type of q and $\forall x.((p\ x) \longleftarrow (q\ x))$ is a logical consequence of $\mathcal{B}$.

Of course, this definition could be given for any predicates, not just standard ones, but that generality is not needed here. Note that, since the type of p is more general than the type of q, it follows that $\forall x.((p\ x) \longleftarrow (q\ x))$ is indeed a term.

Proposition 4.5.1. *The relation* $\Leftarrow$ *is a preorder on* $\mathbf{S}$.

Proof. Let $p, q, r \in \mathbf{S}$. Clearly, $p \Leftarrow p$. Also if $p \Leftarrow q$ and $q \Leftarrow r$, then $p \Leftarrow r$, since $\longleftarrow$ is transitive. Thus $\Leftarrow$ is a preorder. □

The relation $\Leftarrow$ is called the *implication preorder.*

In the top-down construction of predicates, it is crucial to ensure that if a standard predicate q is obtained by a predicate derivation step from a standard predicate p, then $p \Leftarrow q$ holds. Sufficient conditions for this are now given.

Definition 4.5.2. A standard predicate p is *monotone* with respect to a predicate rewrite system $\rightarrowtail$ if, for every disjoint set $\{r_i\}_{i=1}^n$ of redexes in p with respect to $\rightarrowtail$ and for every sets $\{s_i\}_{i=1}^n$ and $\{s_i'\}_{i=1}^n$ of standard predicates such that $r_i \Leftarrow s_i \Leftarrow s_i'$ $(i = 1, \ldots, n)$ and $p[r_1/s_1, \ldots, r_n/s_n]$ and $p[r_1/s_1', \ldots, r_n/s_n']$ are standard predicates, it follows that $p[r_1/s_1, \ldots, r_n/s_n]$ $\Leftarrow p[r_1/s_1', \ldots, r_n/s_n']$.

If a standard predicate contains no redexes with respect to a predicate rewrite system $\rightarrowtail$, then it is monotone with respect to $\rightarrowtail$. If p is a standard predicate whose only redex with respect to a predicate rewrite system $\rightarrowtail$ is a suffix of p, then p is monotone with respect to $\rightarrowtail$.

Example 4.5.1. If $p_1, \ldots, p_n$ are standard predicates that are monotone with respect to a predicate rewrite system, then so are $\wedge_n\ p_1 \ldots p_n$ and $setExists_n\ p_1 \ldots p_n$.

Example 4.5.2. If p is a standard predicate that is monotone with respect to a predicate rewrite system, then so is $(domCard\ p) \circ (> N)$.

One could conjecture that for a standard predicate to be monotone it suffices to consider one redex at a time. It turns out that this is not the case.

Example 4.5.3. Consider the transformation

$$f : (a \to \Omega) \to (a \to \Omega) \to a \to \Omega$$

defined by

$$f\ p\ q = \begin{cases} \mathit{bottom} & \text{if either } p \Leftarrow q \text{ or } q \Leftarrow p \\ \mathit{top} & \text{otherwise.} \end{cases}$$

Let M be a unary type constructor and r and s predicates on M such that $r \not\Leftarrow s$ and $s \not\Leftarrow r$. (One can obtain this by letting M be the type of a two element domain.) Let $\rightarrowtail$ be the predicate rewrite system

$$\mathit{top} \rightarrowtail \mathit{bottom}$$
$$\mathit{top} \rightarrowtail r$$
$$\mathit{top} \rightarrowtail s.$$

Then $f\ \mathit{top}\ \mathit{top}$ is separately monotone in each of its redexes. However, $f\ \mathit{top}\ \mathit{top} \not\Leftarrow f\ r\ s$, and thus $f\ \mathit{top}\ \mathit{top}$ is not monotone.

Definition 4.5.3. A predicate rewrite system $\rightarrowtail$ is *monotone* if the following conditions are satisfied.

1. $p \rightarrowtail q$ implies $p \Leftarrow q$.
2. $p \rightarrowtail q$ implies q is monotone with respect to $\rightarrowtail$.

Example 4.5.4. The predicate rewrite system of Example 4.4.2 is monotone. It is clear that each rewrite $p \rightarrowtail q$ satisfies $p \Leftarrow q$. Furthermore, the body of each rewrite is monotone. For all but the first rewrite, the body contains no redex and is therefore monotone. For the first rewrite, it suffices to note that $\mathit{projMake} \circ \mathit{top}$, $\mathit{projNumProngs} \circ \mathit{top}$, and so on, are all monotone with respect to the predicate rewrite system since they have just one redex that is a suffix and therefore the body is monotone by the earlier remarks about $\wedge_n$ and $\mathit{setExists}_n$.

The monotone property of standard predicates is preserved under predicate derivation steps for monotone predicate rewrite systems.

Proposition 4.5.2. *Let $\rightarrowtail$ be a monotone predicate rewrite system and p a monotone predicate with respect to $\rightarrowtail$. Suppose that q is obtained by a predicate derivation step from p using $\rightarrowtail$. Then the following hold.*

1. $p \Leftarrow q$.
2. *q is monotone with respect to $\rightarrowtail$.*

Proof. 1. Let r be the redex selected in p and $r \rightarrowtail b$ the rewrite employed in the predicate derivation step. Then $r \Leftarrow b$. Since p is monotone, it follows that $p[r/r] \Leftarrow p[r/b]$. That is, $p \Leftarrow q$.

2. Let $\{r_j\}_{j=1}^m$ $(m \geq 1)$ be a disjoint set of redexes in q and $\{s_j\}_{j=1}^m$ and $\{s'_j\}_{j=1}^m$ be sets of standard predicates such that $r_j \Leftarrow s_j \Leftarrow s'_j$ $(j = 1, \ldots, m)$ and $q[r_1/s_1, \ldots, r_m/s_m]$ and $q[r_1/s'_1, \ldots, r_m/s'_m]$ are standard predicates. By Proposition 4.4.3, each r_j is either a redex in p or a redex in b. Let B denote the subset of $\{1, \ldots, m\}$ such that r_j is a redex in b iff $j \in B$. Suppose $B = \{j_1, \ldots, j_k\}$, for some $k \geq 0$, and $\{1, \ldots, m\} \backslash B = \{l_1, \ldots, l_v\}$, for some $v \geq 0$. Since b is monotone and both $b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}]$ and $b[r_{j_1}/s'_{j_1}, \ldots, r_{j_k}/s'_{j_k}]$ are standard predicates, it follows that

$$b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}] \Leftarrow b[r_{j_1}/s'_{j_1}, \ldots, r_{j_k}/s'_{j_k}].$$

Note also that $r \Leftarrow b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}]$, using the facts that $r \Leftarrow b$ and $b \Leftarrow b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}]$, since b is monotone. Consider the disjoint set $\{r, r_{l_1}, \ldots, r_{l_v}\}$ of redexes in p. Since

$$p[r/b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}], r_{l_1}/s_{l_1}, \ldots, r_{l_v}/s_{l_v}] = q[r_1/s_1, \ldots, r_m/s_m]$$

and

$$p[r/b[r_{j_1}/s'_{j_1}, \ldots, r_{j_k}/s'_{j_k}], r_{l_1}/s'_{l_1}, \ldots, r_{l_v}/s'_{l_v}] = q[r_1/s'_1, \ldots, r_m/s'_m],$$

each of the left hand sides is a standard predicate. It follows that

$$p[r/b[r_{j_1}/s_{j_1}, \ldots, r_{j_k}/s_{j_k}], r_{l_1}/s_{l_1}, \ldots, r_{l_v}/s_{l_v}] \Leftarrow \\ p[r/b[r_{j_1}/s'_{j_1}, \ldots, r_{j_k}/s'_{j_k}], r_{l_1}/s'_{l_1}, \ldots, r_{l_v}/s'_{l_v}],$$

since p is monotone. Consequently

$$q[r_1/s_1, \ldots, r_m/s_m] \Leftarrow q[r_1/s'_1, \ldots, r_m/s'_m].$$

Thus q is monotone. □

The last result extends easily to predicate derivations.

Proposition 4.5.3. *Let $\rightarrowtail$ be a monotone predicate rewrite system and $\langle p_0, p_1, \ldots, p_n \rangle$ a predicate derivation using $\rightarrowtail$ such that the initial predicate is monotone with respect to $\rightarrowtail$. Then the following hold.*

1. $p_0 \Leftarrow p_1 \Leftarrow \cdots \Leftarrow p_n$.
2. *p_i is monotone with respect to $\rightarrowtail$, for $i = 1, \ldots, n$.*

Proof. Each part can be proved by a straightforward induction argument on the length of the derivation, using Proposition 4.5.2. □

Example 4.5.5. Consider the predicate derivation given in Example 4.4.2 again. The initial predicate *top* is monotone with respect to the predicate rewrite system that is also monotone. Consequently,

$$\begin{aligned}
&top\\
&\Leftarrow setExists_1\ (\wedge_4\ (projMake \circ top)\ (projNumProngs \circ top)\\
&\qquad (projLength \circ top)\ (projWidth \circ top))\\
&\Leftarrow setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad (projLength \circ top)\ (projWidth \circ top))\\
&\Leftarrow setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad (projLength \circ (= Medium))\ (projWidth \circ top))\\
&\Leftarrow setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad)))\\
&\Leftarrow setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ (= 6))\\
&\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad))).
\end{aligned}$$

Proposition 4.5.3 does not hold if the second condition of Definition 4.4.1 is dropped.

Example 4.5.6. Consider the transformations

$$f : (a \to \Omega) \to a \to \Omega$$

defined by

$$f\ p = top,$$

and

$$\sim : (a \to \Omega) \to a \to \Omega$$

defined by

$$\sim\ p = \lambda x.\neg(p\ x),$$

where a is a parameter. Let $\rightarrowtail$ be the relation

$$\begin{aligned}
&top \rightarrowtail bottom\\
&f\ bottom \rightarrowtail bottom.
\end{aligned}$$

Note that $\rightarrowtail$ is monotone and $\sim f\ top$ is monotone with respect to $\rightarrowtail$. Then

$$\begin{aligned}
&\sim f\ top\\
&\sim f\ bottom\\
&\sim bottom
\end{aligned}$$

is a predicate derivation. However, while $\sim f\ top \Leftarrow\ \sim f\ bottom$, it is not true that $\sim f\ bottom \Leftarrow\ \sim bottom$ and, furthermore, $\sim f\ top \not\Leftarrow\ \sim bottom$. Also $\sim f\ bottom$ is not monotone. The problem is that $\rightarrowtail$ is not a predicate rewrite system because it does not satisfy the second condition of Definition 4.4.1.

In the next section, it will become clear that it is useful to 'regularise' predicate rewrite systems. This concept is introduced now and related to the monotone property.

Definition 4.5.4. A predicate rewrite system $\rightarrowtail$ is *regular* if, for each $p \rightarrowtail q$, $p \in \mathbf{R}$ and $q \in \mathbf{R}$.

Definition 4.5.5. Let $\rightarrowtail$ be a predicate rewrite system. Then the *regularisation* of $\rightarrowtail$ is the finite relation $\overline{\rightarrowtail}$ on $\mathbf{R}$ defined by $p \overline{\rightarrowtail} q$ iff there exist p' and q' such that $p = \overline{p'}$, $q = \overline{q'}$, and $p' \rightarrowtail q'$.

The regularisation of a predicate rewrite system naturally depends upon the choice of total order $<$ on the transformations. In practice, the regularisation of a predicate rewrite system is again a predicate rewrite system (for example, in the case when the heads of the rewrites are standard predicates containing a single transformation). However, in sufficiently bizarre circumstances, this may not be the case, since Condition 2 of the definition of predicate rewrite system may not be satisfied.

Example 4.5.7. Consider the predicate rewrite system $\rightarrowtail$ given by

$$(f\ (g\ a\ b)) \rightarrowtail (g\ b\ a),$$

where f, g, a and b have suitable signatures and g is symmetric. Consider any regularisation based on an order on transformations for which $a < b$. Then the regularisation $\overline{\rightarrowtail}$ is

$$(f\ (g\ a\ b)) \overline{\rightarrowtail} (g\ a\ b),$$

which is not a predicate rewrite system.

If the regularisation of a predicate rewrite system is a predicate rewrite system, then it is clearly regular.

Example 4.5.8. Consider again the predicate rewrite system of Example 4.4.2. Suppose that the total order $<$ is the lexicographic order, so that $projLength < projMake < projNumProngs < projWidth$. Then the regularisation of the predicate rewrite system includes the rewrite

$$\begin{aligned} top \rightarrowtail setExists_1\ (\wedge_4\ &(projLength \circ top)\ (projMake \circ top) \\ &(projNumProngs \circ top)\ (projWidth \circ top)). \end{aligned}$$

The remaining rewrites are unchanged by the regularisation.

Proposition 4.5.4. *Suppose that* $\rightarrowtail$ *is a predicate rewrite system such that the head of each rewrite is a single transformation. Then the following hold.*

1. *The regularisation* $\overline{\rightarrowtail}$ *of* $\rightarrowtail$ *is a predicate rewrite system.*
2. *If a standard predicate* p *is monotone with respect to* $\rightarrowtail$*, then* $\overline{p}$ *is monotone with respect to* $\overline{\rightarrowtail}$*.*
3. *If* $\rightarrowtail$ *is monotone, then* $\overline{\rightarrowtail}$ *is monotone.*

Proof. The proof is straightforward. □

Example 4.5.9. The predicate rewrite systems of Examples 4.4.2 and 4.6.2 are monotone and hence any regularisation of them is monotone.

Checking that a predicate rewrite system is monotone is undecidable and therefore is left to the user. However, with a little practice, it is easy to check that a particular rewrite system satisfies this condition. Not only that, the condition is a weak one; it is natural for predicate rewrite systems in practical applications to be monotone. Furthermore, Proposition 4.5.4 shows that in the usual case when the head of each rewrite is a standard predicate consisting of a single transformation, the regularisation of a monotone predicate rewrite system is a predicate rewrite system that is monotone. Thus an implementation has the freedom to work with the regularisation, if this is convenient. This fact will be exploited in the next section.

4.6 Efficient Construction of Predicates

To efficiently construct predicates in many practical applications, some care is necessary. Consider the following illustrative examples.

Example 4.6.1. Consider the predicate rewrite system of Example 4.4.2. Corresponding to the different choices of redexes, there are 4! distinct predicate derivations with initial predicate *top* that have

$$\begin{aligned}&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ (= 6))\\ &\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad)))\end{aligned}$$

as the final predicate. Clearly it would be preferable to construct only one of these derivations.

Example 4.6.2. Consider the following predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail setExists_1\ (\wedge_4\ top\ top\ top\ top)\\
&top \rightarrowtail projMake \circ (= Abloy)\\
&top \rightarrowtail projMake \circ (= Chubb)\\
&top \rightarrowtail projMake \circ (= Rubo)\\
&top \rightarrowtail projMake \circ (= Yale)\\
&top \rightarrowtail projNumProngs \circ (= 2)\\
&top \rightarrowtail projNumProngs \circ (= 3)\\
&top \rightarrowtail projNumProngs \circ (= 4)\\
&top \rightarrowtail projNumProngs \circ (= 5)\\
&top \rightarrowtail projNumProngs \circ (= 6)\\
&top \rightarrowtail projLength \circ (= Short)\\
&top \rightarrowtail projLength \circ (= Medium)\\
&top \rightarrowtail projLength \circ (= Long)\\
&top \rightarrowtail projWidth \circ (= Narrow)\\
&top \rightarrowtail projWidth \circ (= Normal)\\
&top \rightarrowtail projWidth \circ (= Broad).
\end{aligned}
$$

Then the standard predicates

$$
\begin{aligned}
&setExists_1\ (\wedge_4\ (projNumProngs \circ (= 6))\ (projMake \circ (= Abloy))\\
&\qquad\qquad (projLength \circ (= Medium))\ (projWidth \circ (= Broad)))
\end{aligned}
$$

and

$$
\begin{aligned}
&setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ (= 6))\\
&\qquad\qquad (projWidth \circ (= Broad))\ (projLength \circ (= Medium)))
\end{aligned}
$$

can both be obtained as the final predicates of derivations starting from the initial predicate *top*. But these two standard predicates are logically equivalent. There are 4! such logically equivalent predicates altogether corresponding to the various orderings of the arguments to $\wedge_4$. Clearly, it would be preferable to only construct one of these predicates.

Thus a modified form of predicate derivation is introduced in which the selection of redexes is restricted and also non-regular standard predicates are discarded yet, under mild conditions, all the (non-equivalent) predicates previously obtained can still be obtained with the modified derivation.

I first address the restricted form of redex selection. Note that occurrences (of subterms) can be totally ordered by the lexicographic ordering, denoted by $\leq$. The strict lexicographic ordering is denoted by $<$.

Definition 4.6.1. Let $\langle p_0, p_1, \ldots, p_n \rangle$ be a predicate derivation, where o_i is the occurrence of the redex selected in p_i, for $i = 0, \ldots, n-1$. Then the predicate derivation is said to be *LR* if $o_{i-1} \leq o_i$, for $i = 1, \ldots, n-1$.

LR stands for 'left-to-right'. The intuitive idea behind the definition is that a redex in the predicate derivation is always 'at or to the right of' the immediately preceding selected redex. Hence the selection of redexes proceeds left-to-right.

The first task is to show that any final predicate in a derivation is the final predicate in an LR derivation. To prove that result, a preliminary result concerning the switching of redexes is needed.

Proposition 4.6.1. *Let $\rightarrowtail$ be a predicate rewrite system. Suppose that p_1, p_2, and p_3 are standard predicates such that p_2 is obtained by a derivation step from p_1 using $r_1 \rightarrowtail b_1$ and p_3 is obtained by a derivation step from p_2 using $r_2 \rightarrowtail b_2$. Suppose that the redexes r_1 and r_2 both occur in p_1 and are disjoint. Then there is a standard predicate p'_2 such that p'_2 is obtained by a derivation step from p_1 using $r_2 \rightarrowtail b_2$ and p_3 is obtained by a derivation step from p'_2 using $r_1 \rightarrowtail b_1$.*

Intuitively, the order of selection of r_1 and r_2 can be reversed.

Proof. Since r_1 and r_2 are disjoint in p_1, it is possible to consider reversing the order of selection of redexes. First, I show that the positional type of r_2 in p_1 is more general than the positional type of r_2 in p_2. Let ϱ_1 be the type of r_1 and β_1 be the type of b_1. Since $r_1 \rightarrowtail b_1$, there exists a type substitution ξ such that $\varrho_1\xi = \beta_1$. Now let q be $p_1[r_1/x_1, r_2/x_2]$, where x_1 and x_2 are distinct variables, and δ_i be the relative type of x_i in q, for $i = 1, 2$. Since $q\{x_1/r_1\}$ is a term, $\delta_1 = \varrho_1$ has an mgu θ, say, by Proposition 2.5.4. Similarly, since $q\{x_1/b_1\}$ is a term, $\delta_1 = \beta_1$ has an mgu φ, say. Furthermore, the positional type of r_2 in p_1 is $\delta_2\theta$, while the positional type of r_2 in p_2 is $\delta_2\varphi$. Now $\delta_1\xi\varphi = \delta_1\varphi = \beta_1\varphi = \varrho_1\xi\varphi$. Thus $\xi\varphi$ is a unifier of $\delta_1 = \varrho_1$. It follows that $\xi\varphi = \theta\pi$, for some type substitution π. Thus $\delta_2\varphi = \delta_2\xi\varphi = \delta_2\theta\pi$ and so the positional type of r_2 in p_1 is more general than the positional type of r_2 in p_2.

It follows from this that r_2 can be selected in p_1 to give p'_2. Furthermore, since p_3 is a term, r_1 can be now selected in p'_2 to give p_3. □

Now the result about the existence of LR derivations can be proved.

Proposition 4.6.2. *Let $\langle p_0, p_1, \ldots, p_n \rangle$ be a predicate derivation with respect to a predicate rewrite system $\rightarrowtail$. Then there exists an LR predicate derivation with respect to $\rightarrowtail$ having initial predicate p_0 and final predicate p_n, and using the same set of rewrites.*

Proof. The proof is by induction on the length n of the derivation. If $n = 0$, the result is obvious.

Let $\langle p_0, p_1, \ldots, p_{n+1} \rangle$ be a derivation of length $n+1$ and suppose the result holds for derivations of length n. Thus there is an LR predicate derivation $\langle p'_0, p'_1, \ldots, p'_n \rangle$ with respect to $\rightarrowtail$ such that $p'_0 = p_0$ and $p'_n = p_n$, and using

the same set of rewrites as the first n steps of the given derivation. Suppose that the redex selected in p'_i is r_i at occurrence o_i, for $i = 0, \ldots, n-1$.

Let the redex selected in the original derivation in p_n be r_n at occurrence o_n. If $o_{n-1} \leq o_n$, then with this same choice of redex the LR derivation of length n extended by this extra step remains LR and the result follows. Otherwise, $o_n < o_{n-1}$. By Proposition 4.4.3, r_n is a redex in p'_{n-1}, and r_{n-1} and r_n are disjoint in p'_{n-1}. Using Proposition 4.6.1, the selection of r_{n-1} and r_n can be switched, so that r_n is selected first in p'_{n-1}, followed by r_{n-1} in the resulting predicate. If the derivation so modified is now LR, the proof is complete. Otherwise, the switching step is repeated until an LR derivation that has the desired properties is obtained. □

Example 4.6.3. Consider the predicate derivation of Example 4.4.2. This derivation is not LR. However, using Proposition 4.6.2, the following LR one with the same initial and final predicates can be obtained.

$$
\begin{array}{l}
top \\
setExists_1\ (\wedge_4\ (projMake \circ top)\ (projNumProngs \circ top) \\
\qquad\qquad\qquad (projLength \circ top)\ (projWidth \circ top)) \\
setExists_1\ (\wedge_4\ (projMake \circ (=\ Abloy))\ (projNumProngs \circ top) \\
\qquad\qquad\qquad (projLength \circ top)\ (projWidth \circ top)) \\
setExists_1\ (\wedge_4\ (projMake \circ (=\ Abloy))\ (projNumProngs \circ (=6)) \\
\qquad\qquad\qquad (projLength \circ top)\ (projWidth \circ top)) \\
setExists_1\ (\wedge_4\ (projMake \circ (=\ Abloy))\ (projNumProngs \circ (=6)) \\
\qquad\qquad (projLength \circ (=\ Medium))\ (projWidth \circ top) \\
setExists_1\ (\wedge_4\ (projMake \circ (=\ Abloy))\ (projNumProngs \circ (=6)) \\
\qquad (projLength \circ (=\ Medium))\ (projWidth \circ (=\ Broad))).
\end{array}
$$

This example addresses the problem raised in Example 4.6.1. Instead of having to deal with the 4! possible derivations, it is only necessary to construct the unique LR one.

Now attention turns to the problems caused by symmetric transformations that are illustrated by Example 4.6.2. The first main result is Proposition 4.6.5 below that is a version of Proposition 4.6.2 for predicate derivations in which each predicate is regular. To prove Proposition 4.6.5, a version of Proposition 4.6.1 for regular predicates is needed and, for that, a condition on predicate rewrite systems is imposed.

Definition 4.6.2. A predicate rewrite system $\rightarrowtail$ is *descending* if there is a strict total order $<$ on transformations such that, for each $r \rightarrowtail b$, it follows that $b \prec r$.

Example 4.6.4. The predicate rewrite systems of Examples 4.4.2 and 4.6.2 are both descending. It is only necessary to choose a total order $<$ on transformations so that *top* is the largest transformation.

Proposition 4.6.3. *Let $\rightarrowtail$ be a descending predicate rewrite system, and p and q standard predicates. Suppose that q is obtained by a predicate derivation step from p using $\rightarrowtail$. Then $q \prec p$.*

Proof. The proof is by induction on the number of transformations in p. Thus suppose p has m transformations and the result holds for predicates containing strictly less than m transformations. Suppose that q is $p[r/b]$, where the rewrite used in the derivation step is $r \rightarrowtail b$. Since $\rightarrowtail$ is descending, it follows that $b \prec r$.

Suppose that p is $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$. The redex r is either a suffix of p or an eligible subterm of p_{i,j_i}, for some i and j_i.

If r is a suffix of p, since $b \prec r$, it follows easily that $p[r/b] \prec p$.

Suppose now that r is an eligible subterm of p_{i,j_i}, for some i and j_i. Note first that, by Proposition 2.5.2, r is a redex in p_{i,j_i} because the positional type of r in p_{i,j_i} is more general than the positional type of r in p. Thus the standard predicate $p_{i,j_i}[r/b]$ can be obtained by a predicate derivation step from p_{i,j_i} using the rewrite $r \rightarrowtail b$. By the induction hypothesis, $p_{i,j_i}[r/b] \prec p_{i,j_i}$. Consequently, $p[r/b] \prec p$. □

Proposition 4.6.4. *Let $\rightarrowtail$ be a descending predicate rewrite system. Suppose that p_1, p_2, and p_3 are regular predicates such that p_2 is obtained by a derivation step from p_1 using $r_1 \rightarrowtail b_1$ and p_3 is obtained by a derivation step from p_2 using $r_2 \rightarrowtail b_2$. Suppose that the redexes r_1 and r_2 both occur in p_1 at occurrences o_1 and o_2, respectively, and $o_2 < o_1$. Then there is a regular predicate p_2' such that p_2' is obtained by a derivation step from p_1 using $r_2 \rightarrowtail b_2$ and p_3 is obtained by a derivation step from p_2' using $r_1 \rightarrowtail b_1$.*

Proof. By Proposition 4.4.3, the redexes r_1 and r_2 are disjoint in p_1. Now Proposition 4.6.1 shows that p_2' (that is, $p_1[r_2/b_2]$) exists and is a standard predicate. Thus it is only necessary to show that p_2' is regular.

Suppose that p_1 has the form $(f_1\ p_{1,1} \dots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$. The redex r_2 is either a suffix of p_1 or an eligible subterm of p_{i,j_i}, for some i and j_i. Since r_1 and r_2 are disjoint and $o_2 < o_1$, it follows that r_2 cannot be a suffix.

The proof is by induction on the number of transformations in p_1. Assume that the result (that is, p_2' is regular) holds for predicates containing fewer than m transformations and suppose that p_1 contains m transformations.

Suppose that r_2 is an eligible subterm of p_{i,j_i}, for some i and j_i. There are three subcases to consider.

Suppose first that r_1 is a subterm of $(f_{i+1}\ p_{i+1,1} \dots p_{i+1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \dots p_{n,k_n})$. Then, since p_1 and p_3 are regular, it follows easily that p_2' is regular.

Suppose next that r_1 and r_2 are both redexes in $f_i\ p_{i,1} \ldots p_{i,k_i}$, for some i, but are not in the same p_{i,j_i}. Thus, say, r_1 is a redex in p_{i,j_1} and r_2 is a redex in p_{i,j_2}, where $j_2 < j_1$. Since p_1 and p_3 are regular, it follows that each of $p_{i,1}, \ldots, p_{i,j_2-1}, p_{i,j_2}[r_2/b_2], p_{i,j_2+1}, \ldots, p_{i,k_i}$ is a regular predicate. If f_i is not symmetric, p'_2 is thus a regular predicate. Otherwise, f_i is symmetric and the proof proceeds as follows. Since $b_2 \prec r_2$, it follows from Proposition 4.6.3 that $p_{i,j_2}[r_2/b_2] \prec p_{i,j_2}$. Thus $p_{i,j_2}[r_2/b_2] \prec p_{i,j_2+1}$. Also $p_{i,j_2-1} \preceq p_{i,j_2}[r_2/b_2]$, since p_3 is regular. Thus $p_{i,1} \preceq \cdots \preceq p_{i,j_2-1} \preceq p_{i,j_2}[r_2/b_2] \prec p_{i,j_2+1} \preceq \cdots \preceq p_{i,k_i}$. Hence p'_2 is a regular predicate.

Suppose finally that both r_1 and r_2 are redexes in p_{i,j_i}, say. Consider the predicate subderivation with initial predicate p_{i,j_i} in which redex r_1 is selected first, followed by redex r_2 in the subsequent predicate. For the first redex, $r_1 \rightarrowtail b_1$ is used and, for the second redex, $r_2 \rightarrowtail b_2$ is used. By the induction hypothesis, $p_{i,j_i}[r_2/b_2]$ is a regular predicate. If f_i is not symmetric, p'_2 is thus a regular predicate. Otherwise, $p_{i,1} \preceq \cdots \preceq p_{i,j_i-1} \prec p_{i,j_i}[r_2/b_2] \prec p_{i,j_i+1} \preceq \cdots \preceq p_{i,k_i}$, by the symmetry of f_i, the regularity of p_1 and p_3, and the facts that $b_1 \prec r_1$, $b_2 \prec r_2$ and $p_{i,j_i-1} \preceq p_{i,j_i}[r_2/b_2, r_1/b_1] \prec p_{i,j_i}[r_2/b_2]$, using Proposition 4.6.3. Hence p'_2 is a regular predicate. $\square$

The condition in Proposition 4.6.4 that the predicate rewrite system be descending cannot be dropped.

Example 4.6.5. Consider the predicate rewrite system

$$p \rightarrowtail q$$
$$q \rightarrowtail p,$$

where $p : M \to \Omega$ and $q : M \to \Omega$, and M is a nullary type constructor. Clearly there does not exist a total order on p and q such that $\rightarrowtail$ is descending. Now suppose that $p < q$. Let $f : (M \to \Omega) \to (M \to \Omega) \to M \to \Omega$ be a symmetric transformation. Then each predicate in the predicate derivation

$$f\ p\ p$$
$$f\ p\ q$$
$$f\ q\ q$$

is regular. However, for the predicate derivation

$$f\ p\ p$$
$$f\ q\ p$$
$$f\ q\ q,$$

in which the order of selection of redexes is reversed, $f\ q\ p$ is not regular. If instead one chooses $q < p$, then a similar counterexample can be constructed using $f\ q\ q$ as the initial predicate.

Definition 4.6.3. A predicate derivation $\langle p_0, p_1, \ldots, p_n \rangle$ is *regular* if $p_i \in \mathbf{R}$, for $i = 0, \ldots, n$.

Proposition 4.6.5. *Let* $\rightarrowtail$ *be a descending predicate rewrite system and* $\langle p_0, p_1, \ldots, p_n \rangle$ *a regular predicate derivation with respect to* $\rightarrowtail$*. Then there exists a regular, LR predicate derivation with respect to* $\rightarrowtail$ *having initial predicate* p_0 *and final predicate* p_n*, and using the same set of rewrites.*

Proof. The proof is by induction on the length n of the derivation. If $n = 0$, the result is obvious.

Let $\langle p_0, p_1, \ldots, p_{n+1} \rangle$ be a derivation of length $n + 1$ and suppose the result holds for derivations of length n. Thus there is a regular, LR predicate derivation $\langle p'_0, p'_1, \ldots, p'_n \rangle$ with respect to $\rightarrowtail$ such that $p'_0 = p_0$ and $p'_n = p_n$, and using the same set of rewrites as the first n steps of the given derivation. Suppose that the redex selected in p'_i is r_i at occurrence o_i, for $i = 0, \ldots, n-1$.

Let the redex selected in the original derivation in p_n be r_n at occurrence o_n. If $o_{n-1} \leq o_n$, then with this same choice of redex the LR derivation of length n extended by this extra step remains LR and the result follows. Otherwise, $o_n < o_{n-1}$. By Proposition 4.4.3, r_n is a redex in p'_{n-1}, and r_{n-1} and r_n are disjoint in p'_{n-1}. Using Proposition 4.6.4, the selection of r_{n-1} and r_n can be switched, so that r_n is selected first in p'_{n-1}, followed by r_{n-1} in the resulting predicate. If the derivation so modified is now LR, the proof is complete. Otherwise, the switching step is repeated until an LR derivation that has the desired properties is obtained. □

The next task is to establish that under certain conditions a predicate derivation can be transformed into one in which each predicate in the derivation is regular.

Definition 4.6.4. A regular predicate p is *switchable* with respect to a predicate rewrite system $\rightarrowtail$ if, for every disjoint set $\{r_i\}_{i=1}^n$ of redexes in p with respect to $\rightarrowtail$, where o_i is the occurrence of r_i, for $i = 1, \ldots, n$, and for every set $\{s_i\}_{i=1}^n$ of regular predicates such that $p[r_1/s_1, \ldots, r_n/s_n]_{o_1,\ldots,o_n}$ is a standard predicate, there exists a disjoint set of occurrences $\{o'_i\}_{i=1}^n$ in p such that r_i also occurs at o'_i, for $i = 1, \ldots, n$, and $p[r_1/s_1, \ldots, r_n/s_n]_{o'_1,\ldots,o'_n}$ is the regularisation of $p[r_1/s_1, \ldots, r_n/s_n]_{o_1,\ldots,o_n}$.

If a regular predicate contains no redexes with respect to a predicate rewrite system $\rightarrowtail$, then it is switchable with respect to $\rightarrowtail$.

Example 4.6.6. Let p be the standard predicate

$$\mathit{setExists}_1\ (\wedge_4\ (\mathit{projMake} \circ \mathit{top})\ (\mathit{projNumProngs} \circ \mathit{top})\ (\mathit{projLength} \circ \mathit{top})\ (\mathit{projWidth} \circ \mathit{top})).$$

Suppose $<$ is defined so that *projMake* $<$ *projNumProngs* $<$ *projLength* $<$ *projWidth*. Then p is regular. Furthermore, p is switchable with respect to

the predicate rewrite system of Example 4.4.2. In fact, for this example, the o'_i in the definition of switchable are the same as the o_i. The reason is that the ordering on *projMake*, *projNumProngs*, and so on, forces regularity no matter what the *top*s are replaced by.

Example 4.6.7. The standard predicate $setExists_1$ ($\wedge_4$ *top top top top*) is obviously switchable with respect to the predicate rewrite system of Example 4.6.2.

The main cases of interest for switchable predicates are illustrated by the last two examples: either the required regularity property in the definition is forced or there is complete freedom to reorder in order to achieve it.

Example 4.6.8. Let $f : (\varrho \to \Omega) \to (\varrho \to \Omega) \to \mu \to \sigma$ be a symmetric transformation, and $p : \varrho \to \Omega$ and $q : \varrho \to \Omega$. Suppose that $p < q < top$. Then $f\ q\ top$ is regular. Now let $\rightarrowtail$ be the predicate rewrite system

$$top \rightarrowtail p.$$

Then $f\ q\ top$ is not switchable with respect to $\rightarrowtail$, since $(f\ q\ top)[top/p]$ is not regular and there is only one occurrence of *top*.

Definition 4.6.5. A regular predicate rewrite system $\rightarrowtail$ is *switchable* if, for each $p \rightarrowtail q$, q is switchable with respect to $\rightarrowtail$.

Example 4.6.9. The predicate rewrite system of Example 4.4.2 is switchable (for any order $<$ such that $projMake < projNumProngs < projLength < projWidth$). The predicate rewrite system of Example 4.6.2 is switchable (for any choice of $<$).

Proposition 4.6.6. *Let $\rightarrowtail$ be a descending, switchable predicate rewrite system and p_0 a switchable predicate. If $\langle p_0, p_1, \ldots, p_n \rangle$ is a predicate derivation with respect to $\rightarrowtail$, then there exists a regular predicate derivation with respect to $\rightarrowtail$ having initial predicate p_0 and final predicate $\overline{p_n}$.*

Proof. The proof is by induction on the length of the derivation. If the length is 0, then the result is obvious.

Now suppose that the result holds for derivations of length $< m$. Let $\langle p_0, p_1, \ldots, p_m \rangle$ be a derivation of length m. In this derivation, some selected redexes originate in p_0 and some in the bodies of rewrites that are used. Let $\{r_1, \ldots, r_n\}$ be the set of selected redexes that originate in p_0, where r_i is at occurrence o_i in p_0, for $i = 1, \ldots, n$. According to Proposition 4.4.3, this set of redexes is disjoint. Suppose the rewrite used at redex r_i is $r_i \rightarrowtail b_i$, for $i = 1, \ldots, n$. Consider the predicate subderivation with initial predicate b_i, for $i = 1, \ldots, n$. Each of these is a predicate derivation, by Proposition 4.4.5. Also the initial predicate of each is a switchable predicate and each has length $< m$. By the induction hypothesis, there is a regular derivation with initial predicate b_i

and where the final predicate s_i is the regularisation of the final predicate of the original derivation, for $i = 1, \ldots, n$. Since p_0 is switchable, there exists a disjoint set $\{o'_i\}_{i=1}^n$ of occurrences in p_0 such that $p_0[r_1/s_1, \ldots, r_n/s_n]_{o'_1,\ldots,o'_n}$ is the regularisation of $p_0[r_1/s_1, \ldots, r_n/s_n]_{o_1,\ldots,o_n}$.

A new derivation with initial predicate p_0 is now constructed as follows. Choose i_1 such that o'_{i_1} is the smallest of $o'_1, \ldots, o'_n$. In the new derivation, let the first selected redex be r_{i_1} at o'_{i_1} and continue with the selection of redexes in the regular derivation that has initial predicate b_{i_1}. Next choose the second smallest o'_{i_2} of $o'_1, \ldots, o'_n$ and continue with the redex r_{i_2} at o'_2 and the redexes in the regular derivation that has initial predicate b_{i_2}, and so on. One can show that this is indeed a predicate derivation because $p_0[r_1/s_1, \ldots, r_n/s_n]_{o'_1,\ldots,o'_n}$ is a term. Furthermore, the predicate derivation so constructed is regular, since each subderivation is regular, the final predicate $p_0[r_1/s_1, \ldots, r_n/s_n]_{o'_1,\ldots,o'_n}$ is regular, and $\rightarrowtail$ is descending. □

Proposition 4.6.7. *Let $\rightarrowtail$ be a predicate rewrite system whose regularisation $\overline{\rightarrowtail}$ is a descending, switchable predicate rewrite system, and p_0 a standard predicate whose regularisation is switchable. If $\langle p_0, p_1, \ldots, p_n \rangle$ is a predicate derivation with respect to $\rightarrowtail$, then there exists a regular predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$.*

Proof. The first step is to show that there exists a predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate p'_n, where p_n and p'_n have the same regularisation and such that if s is a standard predicate which is a subterm at occurrence o of p_n that was introduced either as a subterm of p_0 or one of the bodies of the rewrites used in the derivation and s' is the standard predicate that is the subterm at occurrence o' of p'_n such that $o^t = o'^t$, then s' is regular. The proof is by induction on the length of the derivation $\langle p_0, p_1, \ldots, p_n \rangle$. If the length is 0, then the result is obvious.

Next suppose the result holds for derivations of length $< n$. Let $\langle p_0, p_1, \ldots, p_n \rangle$ be a derivation of length n. By the induction hypothesis, there is a derivation $\langle \overline{p_0}, p'_1, \ldots, p'_{n-1} \rangle$ with respect to $\overline{\rightarrowtail}$ such that p_{n-1} and p'_{n-1} have the same regularisation and the property about subterms of p_{n-1} is satisfied. Let r be the redex selected in p_{n-1}, where the occurrence of r in p_{n-1} is o, and $r \rightarrowtail b$ be the rewrite used. Then there is a subterm r' at occurrence o' in p'_{n-1} such that $o^t = o'^t$, and the positional type of r' in p'_{n-1} is the same as the positional type of r in p_{n-1}, by Proposition 4.3.4. Since r was either introduced as a subterm of p_0 or one of the bodies of the rewrites used in the derivation, r' is a regular predicate, by the property about subterms of p_{n-1}. Hence $r' = \overline{r}$ and r' is a redex. The rewrite used for this redex is $\overline{r} \rightarrowtail \overline{b}$. By Proposition 4.3.5, the regularisations of $p_{n-1}[r/b]_o$ and $p'_{n-1}[\overline{r}/\overline{b}]_{o'}$ are the same. It only remains to show that p_n satisfies the property about subterms. But this follows since standard predicates that are subterms of p_n introduced by b have the property that the corresponding subterms in p'_n are subterms of $\overline{b}$ and therefore are regular. This completes the induction argument.

Since $\overline{\rightarrowtail}$ is a descending, switchable predicate rewrite system and $\overline{p_0}$ is a switchable predicate, by Proposition 4.6.6, there exists a regular predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p'_n}$. Since p_n and p'_n have the same regularisation, the result follows. □

Now an important completeness result can be proved which shows that, under weak conditions, an implementation can employ the LR selection of redexes and discard non-regular predicates in predicate derivations and yet still generate all the (regularisations of) standard predicates that can be generated by unrestricted predicate derivations. Since employing the LR selection of redexes and discarding non-regular predicates avoids considerable redundancy in many common situations, the efficiency of predicate generation is thereby greatly improved.

Proposition 4.6.8. *Let $\rightarrowtail$ be a predicate rewrite system whose regularisation $\overline{\rightarrowtail}$ is a descending, switchable predicate rewrite system, and p_0 a standard predicate whose regularisation is switchable. If $\langle p_0, p_1, \ldots, p_n \rangle$ is a predicate derivation with respect to $\rightarrowtail$, then there exists a regular, LR predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$.*

Proof. By Proposition 4.6.7, there is a regular predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$. Then, by Proposition 4.6.5, there is a regular, LR predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$. □

The final task is to give a sufficient condition to ensure the uniqueness of predicate derivations with the same initial and final predicates.

Definition 4.6.6. A predicate rewrite system $\rightarrowtail$ is *separable* if, for each $p \rightarrowtail q_1$ and $p \rightarrowtail q_2$, whenever there exist predicate derivations with initial predicates q_1 and q_2 having the same final predicate, it follows that $q_1 = q_2$.

Proposition 4.6.9. *If two LR predicate derivations with respect to a separable, descending predicate rewrite system have the same initial and final predicates, then the predicate derivations are identical.*

Proof. Let there be given two LR predicate derivations with respect to a separable, descending predicate rewrite system $\rightarrowtail$ that have the same initial and final predicates. The proof is by induction on the maximum k of the lengths of the two predicate derivations. If $k = 0$, then the result is obvious.

Suppose now that the result is true for predicate derivations of length $< k$ and $D \equiv \langle p_0, p_1, \ldots, p_n \rangle$ and $D' \equiv \langle p'_0, p'_1, \ldots, p'_m \rangle$ be two LR predicate derivations such that $p_0 = p'_0$, $p_n = p'_m$, and $\max(n, m) = k$. Since $k > 0$, at least one of D or D' has non-zero length. If the other has zero length, then the non-zero length derivation has identical initial and final predicates which contradicts Proposition 4.6.3. Thus each predicate derivation has non-zero length. Consider the selected redex in the initial predicates of D and D'.

Since each predicate derivation is LR and the final predicate of each is the same, the redex selected (including its occurrence, of course) in each of the initial predicates must be identical. Let this redex be r and the respective rewrites used at the first step be $r \rightarrowtail q$ in D and $r \rightarrowtail q'$ in D'. Consider now the predicate subderivation of D with initial predicate q that is a subterm of p_1 and the predicate subderivation of D' with initial predicate q' that is a subterm of p'_1. The final predicates of these subderivations, being subterms at the same occurrence of the common final predicate of D and D', are identical. Since $\rightarrowtail$ is separable, it follows that $q = q'$. Hence the second predicates, p_1 in D and p'_1 in D', are identical. The induction hypothesis can now be applied to the remainder of D and D' to obtain the result. □

The condition in Proposition 4.6.9 that the predicate rewrite system be separable cannot be dropped.

Example 4.6.10. Consider the descending predicate rewrite system

$$\begin{aligned} &top \rightarrowtail p \\ &top \rightarrowtail q \\ &q \rightarrowtail p. \end{aligned}$$

Then $\langle top, p\rangle$, and $\langle top, q, p\rangle$ are two distinct LR predicate derivations with the same initial and final predicates.

The condition in Proposition 4.6.9 that the predicate rewrite system be descending cannot be dropped.

Example 4.6.11. Consider the separable predicate rewrite system

$$\begin{aligned} &p \rightarrowtail q \\ &q \rightarrowtail p. \end{aligned}$$

Then $\langle p\rangle$ and $\langle p, q, p\rangle$ are two distinct LR predicate derivations with the same initial and final predicates.

Proposition 4.6.10. *Let $\rightarrowtail$ be a predicate rewrite system whose regularisation $\overline{\rightarrowtail}$ is a separable, descending, switchable predicate rewrite system, and p_0 a standard predicate whose regularisation is switchable. If $\langle p_0, p_1, \ldots, p_n\rangle$ is a predicate derivation with respect to $\rightarrowtail$, then there exists a unique regular, LR predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$.*

Proof. According to Proposition 4.6.8, there exists a regular, LR predicate derivation with respect to $\overline{\rightarrowtail}$ having initial predicate $\overline{p_0}$ and final predicate $\overline{p_n}$. By Proposition 4.6.9, this predicate derivation is unique. □

Example 4.6.12. The predicate rewrite systems of Examples 4.4.2 and 4.6.2 are separable and switchable for any choice of regularisation. Also, for orderings $<$ on transformations for which *top* is declared to be the greatest transformation, the corresponding regularisations are descending. Lastly, in the usual case that the initial predicate is *top*, it follows that all regularisations of the initial predicate are switchable. Thus the conditions of Proposition 4.6.10 are satisfied by these predicate rewrite systems (where *top* is the initial predicate) and there is a guarantee about the existence and uniqueness of regular, LR predicate derivations.

The important results of this section are now summarised. First, from the user perspective, the view is that the hypothesis language is given by all predicates (the so-called *expected* predicates) that are the final predicates of predicate derivations (usually with *top* as the initial predicate), with no restrictions on the selection of redexes. This provides a simple and direct understanding of the search space of expected predicates.

Now it was noted above that a direct implementation of this view leads to redundancy in many applications because there may be several derivation paths to the same predicate and also several equivalent forms of the same predicate may be generated because of the presence of symmetric transformations, especially $\wedge_n$. To combat this, two ideas were introduced: the LR selection rule for the first problem and regular predicates for the second. Thus an efficient implementation of these ideas actually works with a regularisation of the predicate rewrite system, discards non-regular predicates, and employs the LR selection rule. An algorithm for generating a search space of regular predicates according to this implementation is given in Fig. 4.5. In this figure, the phrase 'LR redex' means the redex is selected according to the LR selection rule, that is, a redex selected in a predicate must be at or to the right of the redex selected in the parent of the predicate.

Two important questions then arise.

1. Is such an implementation complete, that is, are (the regularisations of) all the expected predicates actually generated by regular, LR derivations?
2. Is each (regularisation of an) expected predicate generated exactly once?

Proposition 4.6.10 answers these questions. The condition in this proposition is that there be a regularisation of the predicate rewrite system that is separable, descending, and switchable. While this is a reasonably technical condition, it is actually quite weak and likely to be satisfied by predicate rewrite systems in practical applications.

Whether there is a regularisation of the predicate rewrite system that is descending is a decidable condition (assuming there are only finitely many transformations) that can be checked by the implementation. So this much is easy. The other requirements, separable and switchable, are not decidable and so checking that they are satisfied is left to the user. However, with a little experience, it is straightforward to check whether any regularisation of a

```
function Enumerate2(↣, p_0) returns the set of regular predicates in a
                                search space rooted at p̄_0 using ↣̄;

input: ↣, a predicate rewrite system;
       p_0, a predicate;

predicates := {};
openList := [p̄_0];
while openList ≠ [] do
    p := head(openList);
    openList := tail(openList);
    predicates := predicates ∪ {p};
    for each LR redex r via r ↣̄ b, for some b, in p do
        q := p[r/b];
        if q is regular then openList := openList ++ [q];
return predicates;
```

Fig. 4.5. Algorithm for enumerating regular predicates

particular predicate rewrite system is separable and switchable. (See Exercise 4.13.) Furthermore, one has to have a rather strange predicate rewrite system for its regularisations not to be both separable and switchable. Assuming that the condition of Proposition 4.6.10 is satisfied, there is a guarantee that (the regularisation of) each expected predicate is generated by a regular, LR predicate derivation and, indeed, only generated once. Even in the case when the condition is not satisfied, the implementation is sound; that is, only (regularisations of) expected predicates are generated.

Bibliographical Notes

The higher-order approach to predicate construction first appeared in [7], and was developed in [8], [55], and [57]. The traditional approach in first-order logic (usually limited to clauses or Horn clauses) is described comprehensively in [70], which also contains extensive references.

Restrictions on the form of the hypothesis language are called language bias. A discussion of various approaches to language bias and, more generally, to declarative bias which includes such restrictions as types and modes is contained in [65]. The idea of using some kind of grammar to specify language bias is an obvious one that many authors have exploited. A good example of this is [13] in which so-called antecedent description grammars are used. These are a kind of first-order relative of the higher-order predicate rewrite systems of this chapter.

In the context of predictive toxicology, domain-specific predicates have been studied, including such things as the possible existence of 2-dimensional substructures like benzene rings and methyl groups, in [43] and [86].

Exercises

4.1 Prove that each prefix and each suffix of a standard predicate is a term.

4.2 Prove that an eligible subterm of a standard predicate is a standard predicate.

4.3 Prove that, if $f : (\varrho \to \Omega) \to \cdots \to (\varrho \to \Omega) \to \mu \to \sigma$ and $f\ p_1 \ldots p_k$ is a term, then $f\ p_{i_1} \ldots p_{i_k}$ is a term, for all permutations i of $\{1, \ldots, k\}$.

4.4 Prove each of the following.

(i) The regularisation of a standard predicate is unique.
(ii) The regularisation of a standard predicate has the same type as the standard predicate.
(iii) The regularisation of a standard predicate is a regular predicate.
(iv) A standard predicate and its regularisation are equivalent.
(v) The regularisation of a regular predicate is itself.
(vi) If two standard predicates have the same regularisation, then they contain the same number of transformations.

4.5 Prove or disprove: a subterm of a regular predicate that is a standard predicate is regular.

4.6 Find a predicate rewrite system $\rightarrowtail$ and a standard predicate p that is monotone with respect to $\rightarrowtail$ such that $(domCard\ p) \circ (< N)$ is not monotone with respect to $\rightarrowtail$.

4.7 A standard predicate p is *weakly monotone* with respect to a predicate rewrite system $\rightarrowtail$ if, for every disjoint set $\{r_i\}_{i=1}^n$ of redexes in p with respect to $\rightarrowtail$ and for every set $\{s_i\}_{i=1}^n$ of standard predicates such that $r_i \Leftarrow s_i$ $(i = 1, \ldots, n)$ and $p[r_1/s_1, \ldots, r_n/s_n]$ is a standard predicate, it follows that $p \Leftarrow p[r_1/s_1, \ldots, r_n/s_n]$.

A predicate rewrite system $\rightarrowtail$ is *weakly monotone* if $p \rightarrowtail q$ implies $p \Leftarrow q$ and $p \rightarrowtail q$ implies q is weakly monotone with respect to $\rightarrowtail$.

(i) Prove that a standard predicate that is monotone with respect to a predicate rewrite system $\rightarrowtail$ is weakly monotone with respect to $\rightarrowtail$.
(ii) Prove that a monotone predicate rewrite system is weakly monotone.
(iii) Let $\rightarrowtail$ be a weakly monotone predicate rewrite system and $p_0, p_1, \ldots, p_n$ a predicate derivation using $\rightarrowtail$ such that the initial predicate is weakly monotone with respect to $\rightarrowtail$. Prove that $p_0 \Leftarrow p_n$.
(iv) Under the conditions of (iii), prove or disprove: $p_0 \Leftarrow p_1 \Leftarrow \cdots \Leftarrow p_n$.
(v) Prove or disprove: if p is a standard predicate that is weakly monotone with respect to a weakly monotone predicate rewrite system $\rightarrowtail$ and r is a redex in p via $r \rightarrowtail s$, then $p[r/s]$ is weakly monotone.

4.8 A *skeleton* is a term of the form

$$(f_1\ x_{1,1} \ldots x_{1,k_1}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n}),$$

where f_i is a transformation of rank k_i, for $i = 1, \ldots, n$, the target of f_n is Ω, and the x_{i,j_i} are distinct variables, for $i = 1, \ldots, n$ and $j_i = 1, \ldots, k_i$.

A variable $x_{r,s}$, for some $r \in \{1, \ldots, n\}$ and some $s \in \{1, \ldots, k_r\}$, is a *monotone argument* of a skeleton

$$(f_1\ x_{1,1} \ldots x_{1,k_1}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n})$$

if, for any standard predicate p of the form

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$$

which is an instance of the skeleton and for any standard predicate c such that $p_{r,s} \Leftarrow c$, it follows that $p \Leftarrow p[p_{r,s}/c]$.

(i) Determine the monotone arguments for the skeletons $(setExists_n\ x_1 \ldots x_n)$ and $(\wedge_n\ x_1 \ldots x_n)$.

(ii) Let $(f_1\ x_{1,1} \ldots x_{1,k_1}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n})$ be a skeleton, f_r a symmetric transformation, and $x_{r,s}$ a monotone argument of the skeleton, for some $r \in \{1, \ldots, n\}$ and some $s \in \{1, \ldots, k_r\}$. Prove that x_{r,j_r} is a monotone argument of the skeleton, for $j_r = 1, \ldots, k_r$.

4.9 Let $\mathcal{S}$ be a set of skeletons together with their monotone arguments, p a standard predicate $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$ and q a standard predicate that has an occurrence as a subterm of p. Then q is in a *monotone position* with respect to $\mathcal{S}$ in p if either

1. q is a suffix $(f_t\ p_{t,1} \ldots p_{t,k_t}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$ of p, for some $t \geq 1$, or
2. there is a skeleton $(f_t\ x_{t,1} \ldots x_{t,k_t}) \circ \cdots \circ (f_n\ x_{n,1} \ldots x_{n,k_n})$, for some $t \geq 1$, having a monotone argument $x_{r,s}$, where $r \in \{t, \ldots, n\}$ and $s \in \{1, \ldots, k_r\}$, such that q is in a monotone position with respect to $\mathcal{S}$ in $p_{r,s}$.

Assume that there is given a set of skeletons together with their monotone arguments. Let $p, q, r \in \mathbf{S}$. Prove the following.

(i) p is in a monotone position in p.

(ii) If q is in a monotone position in p, then q is in a monotone position in $s \circ p$, for any prefix s such that $s \circ p \in \mathbf{S}$.

(iii) If r is in a monotone position in q and q is in a monotone position in p, then r is in a monotone position in p.

4.10 Assume that there is given a set of skeletons together with their monotone arguments. Let $p, q \in \mathbf{S}$, where q has an occurrence as a subterm of p. Prove that if q is in a monotone position in p then, whenever q' is a standard predicate such that $p[q/q']$ is a term and $q \Leftarrow q'$, it follows that $p \Leftarrow p[q/q']$.

4.11 Let p be the predicate defined by

$$vertices \circ (setExists_2\ (\wedge_2\ (proj_1 \circ (= A))\ (proj_2 \circ (= 3)))\ (proj_1 \circ (= B))).$$

Suppose the skeletons are $(setExists_2\ x\ y)$ with x and y being monotone arguments, and $(\wedge_2\ x\ y)$ with x and y being monotone arguments. Determine the set of monotone positions in p.

4.12 Give an algorithm that takes a standard predicate and a set of skeletons together with their monotone arguments as input and returns the set of monotone positions of the predicate.

4.13 Let *Element* be the type of elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

Make the following type synonyms.

$$\begin{aligned}
&I_1 = \Omega \\
&I_a = \Omega \\
&\epsilon_{LUMO} = Float \\
&AtomType = Nat \\
&Charge = Float \\
&Atom = Element \times AtomType \times Charge \\
&Bond = Nat \\
&Structure = Graph\ Atom\ Bond \\
&Molecule = I_1 \times I_a \times \epsilon_{LUMO} \times Structure.
\end{aligned}$$

Consider also the following transformations.

$$\begin{aligned}
&(= \top) : I_1 \to \Omega \\
&(= \bot) : I_1 \to \Omega \\
&(= \top) : I_a \to \Omega \\
&(= \bot) : I_a \to \Omega \\
&(\leq -3.718) : \epsilon_{LUMO} \to \Omega \\
&(\leq -3.368) : \epsilon_{LUMO} \to \Omega \\
&(\leq -3.168) : \epsilon_{LUMO} \to \Omega \\
&(\leq -3.018) : \epsilon_{LUMO} \to \Omega \\
&(\leq -2.668) : \epsilon_{LUMO} \to \Omega \\
&(\leq -2.418) : \epsilon_{LUMO} \to \Omega \\
&(\leq -0.718) : \epsilon_{LUMO} \to \Omega \\
&(\geq -3.418) : \epsilon_{LUMO} \to \Omega
\end{aligned}$$

$(\geq -3.218) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -3.068) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.918) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.618) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -2.368) : \epsilon_{LUMO} \rightarrow \Omega$
$(\geq -0.668) : \epsilon_{LUMO} \rightarrow \Omega$
$(= Br) : Element \rightarrow \Omega$

$\vdots$

$(= S) : Element \rightarrow \Omega$
$(= 1) : AtomType \rightarrow \Omega$
$(= 3) : AtomType \rightarrow \Omega$

$\vdots$

$(= 232) : AtomType \rightarrow \Omega$
$(\geq -0.781) : Charge \rightarrow \Omega$
$(\geq -0.424) : Charge \rightarrow \Omega$
$(\geq -0.067) : Charge \rightarrow \Omega$
$(\geq 0.290) : Charge \rightarrow \Omega$
$(\geq 0.647) : Charge \rightarrow \Omega$
$(\geq -0.424) : Charge \rightarrow \Omega$
$(\geq -0.067) : Charge \rightarrow \Omega$
$(\geq 0.290) : Charge \rightarrow \Omega$
$(\geq 0.647) : Charge \rightarrow \Omega$
$(\geq 1.004) : Charge \rightarrow \Omega$
$(= 1) : Bond \rightarrow \Omega$
$(= 2) : Bond \rightarrow \Omega$
$(= 3) : Bond \rightarrow \Omega$
$(= 4) : Bond \rightarrow \Omega$
$(= 5) : Bond \rightarrow \Omega$
$(= 7) : Bond \rightarrow \Omega$
$(> 0) : Nat \rightarrow \Omega$
$(> 1) : Nat \rightarrow \Omega$
$(> 2) : Nat \rightarrow \Omega$
$(> 3) : Nat \rightarrow \Omega$
$(> 4) : Nat \rightarrow \Omega$
$projI_1 : Molecule \rightarrow I_1$

$$
\begin{aligned}
&projI_a : Molecule \to I_a\\
&proj\epsilon_{LUMO} : Molecule \to \epsilon_{LUMO}\\
&projStructure : Molecule \to Structure\\
&projElement : Atom \to Element\\
&projAtomType : Atom \to AtomType\\
&projCharge : Atom \to Charge\\
&vertices : Structure \to \{Vertex\ Atom\ Bond\}\\
&edges : Structure \to \{Edge\ Atom\ Bond\}\\
&vertex : Vertex\ Atom\ Bond \to Atom\\
&edge : Edge\ Atom\ Bond \to Bond\\
&connects : Edge\ Atom\ Bond \to (Vertex\ Atom\ Bond \to Nat)\\
&domCard : (Vertex\ Atom\ Bond \to \Omega) \to \{Vertex\ Atom\ Bond\} \to Nat\\
&domCard : (Edge\ Atom\ Bond \to \Omega) \to \{Edge\ Atom\ Bond\} \to Nat\\
&msetExists_2 : (Vertex\ Atom\ Bond \to \Omega) \to (Vertex\ Atom\ Bond \to \Omega)\\
&\qquad\qquad \to (Vertex\ Atom\ Bond \to Nat) \to \Omega\\
&\wedge_2 : (Charge \to \Omega) \to (Charge \to \Omega) \to Charge \to \Omega\\
&\wedge_3 : (Atom \to \Omega) \to (Atom \to \Omega) \to (Atom \to \Omega) \to Atom \to \Omega\\
&\wedge_4 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to (Molecule \to \Omega)\\
&\qquad\qquad \to (Molecule \to \Omega) \to Molecule \to \Omega.
\end{aligned}
$$

Let $\rightarrowtail$ be defined as follows.

$$
\begin{aligned}
&top \rightarrowtail \wedge_4\ (projI_1 \circ top)\ (projI_a \circ top)\ (proj\epsilon_{LUMO} \circ top)\\
&\qquad\qquad (projStructure \circ top)\\
&top \rightarrowtail (= \top)\\
&top \rightarrowtail (= \bot)\\
&top \rightarrowtail (\leq -0.718)\\
&(\leq -0.718) \rightarrowtail (\leq -2.418)\\
&\qquad \vdots\\
&(\leq -3.368) \rightarrowtail (\leq -3.718)\\
&top \rightarrowtail (\geq -3.418)\\
&(\geq -3.418) \rightarrowtail (\geq -3.218)\\
&\qquad \vdots\\
&(\geq -2.368) \rightarrowtail (\geq -0.668)\\
&top \rightarrowtail vertices \circ (domCard\ (vertex \circ top)) \circ (> 0)\\
&top \rightarrowtail edges \circ (domCard\ top) \circ (> 0)
\end{aligned}
$$

$$
\begin{aligned}
&top \rightarrowtail \wedge_3\ (projElement \circ top)\ (projAtomType \circ top) \\
&\qquad\qquad\qquad\qquad (\wedge_2\ (projCharge \circ top)\ (projCharge \circ top)) \\
&top \rightarrowtail \wedge_2\ (edge \circ top) \\
&\qquad\qquad (connects \circ (msetExists_2\ (vertex \circ top)\ (vertex \circ top))) \\
&top \rightarrowtail (= Br) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= S) \\
&top \rightarrowtail (= 1) \\
&top \rightarrowtail (= 3) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= 232) \\
&top \rightarrowtail (\geq -0.781) \\
&(\geq -0.781) \rightarrowtail (\geq -0.424) \\
&\qquad\qquad \vdots \\
&(\geq 0.647) \rightarrowtail (\geq 1.004) \\
&top \rightarrowtail (= 1) \\
&\qquad\qquad \vdots \\
&top \rightarrowtail (= 7) \\
&(> 0) \rightarrowtail (> 1) \\
&(> 1) \rightarrowtail (> 2) \\
&(> 2) \rightarrowtail (> 3) \\
&(> 3) \rightarrowtail (> 4).
\end{aligned}
$$

Provide answers, giving detailed reasons, to each of the following questions.

(i) Is $\rightarrowtail$ a predicate rewrite system?
(ii) Is every regularisation of $\rightarrowtail$ a predicate rewrite system?
(iii) Is $\rightarrowtail$ monotone?
(iv) Is every regularisation of $\rightarrowtail$ monotone?
(v) Is there a regularisation of $\rightarrowtail$ that is separable, or descending, or switchable, or all three?
(vi) Is every regularisation of $\rightarrowtail$ separable, or descending, or switchable, or all three?

5. Computation

In this chapter, the paradigm of programming with equational theories is presented. This paradigm provides a suitable computational formalism for the applications to learning, as well as many other applications of the logic.

5.1 Programs as Equational Theories

The first task is to define the formulas that can appear in a program.

Definition 5.1.1. A *statement* is a term of the form $h = b$, where h has the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, each free variable in h occurs exactly once in h, and $b \precsim h$.

The term h is called the *head* and the term b is called the *body* of the statement. The statement is said to be *about* f.

Proposition 5.1.1. *Suppose that h is a term of the form $f\ t_1 \ldots t_n$, $n \geq 0$, for some function f, each free variable in h occurs exactly once in h, and b is a term such that $b \precsim h$. Then $h = b$ is a statement.*

Proof. It is only necessary to show that $h = b$ is a term. Suppose that h has type σ. Then $(=\ h)$ is a term of type $\sigma \to \Omega$. Thus $((=\ h)\ b)$ is a term of type Ω, since $b \precsim h$. □

Example 5.1.1. Consider the function $append : List\ a \times List\ a \times List\ a \to \Omega$. Then the following term is a statement about $append$.

$$\begin{aligned} append\ (u, v, w) &= (u = [] \ \wedge\ v = w) \vee \\ &\qquad \exists r.\exists x.\exists y.(u = r \sharp x \ \wedge\ w = r \sharp y \ \wedge\ append\ (x, v, y)). \end{aligned}$$

The head has type Ω and the free variables u, v, and w have relative type $List\ a$ in the head. The body also has type Ω and its free variables u, v, and w have relative type $List\ a$ in the body. Thus the body is type-weaker than (in fact, type-equivalent to) the head.

Usually, the head and the body of a statement are type-equivalent, but this is not always the case.

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

Example 5.1.2. Consider the statement

$$concat\ []\ x = x$$

about the function $concat : List\ a \to List\ a \to List\ a$. Here h is $concat\ []\ x$ and b is x. Then h has type $List\ a$ and b has type a', for some parameters a and a'. Thus b is type-weaker than h with $\gamma = \{a'/List\ a\}$.

Definition 5.1.2. The *definition* of a function f is the collection of all statements about f, together with the signature for f.

Definition 5.1.3. A *program* is a collection of definitions.

A program is a theory in which each formula is a particular kind of equation, in fact, a statement. Most of the theories of interest in this book are programs.

Now I turn to the computational model for programs. The objective of this computational model is to 'evaluate' terms, where 'evaluate' has the intuitive meaning of 'find the simplest form of'. Thus one would expect the evaluation of $concat\ 1 \sharp 2 \sharp []\ 3 \sharp []$ to result in $1 \sharp 2 \sharp 3\ []$, where $concat$ is the function that concatenates two lists. The evaluation proceeds by a sequence of steps until no more simplification is possible. The primary computational problem for the applications to learning is to evaluate an expression of the form $(p\ t)$, where p is a standard predicate and t is a basic term representing an individual.

Definition 5.1.4. A *redex* of a term t is an occurrence of a subterm of t that is α-equivalent to an instance of the head of a statement.

A redex is *outermost* if it is not a (proper) subterm of another redex.

Definition 5.1.5. Let $\mathfrak{L}$ be the set of terms constructed from the alphabet of a program and $\mathfrak{DS}_{\mathfrak{L}}$ the set of subterms of terms in $\mathfrak{L}$ (distinguished by their occurrences). A *selection rule* S is a function from $\mathfrak{L}$ to the power set of $\mathfrak{DS}_{\mathfrak{L}}$ satisfying the following condition: if t is a term in $\mathfrak{L}$, then $S(t)$ is a subset of the set of outermost redexes in t.

Clearly $S(t)$ is a disjoint set of subterms of t, since the redexes are outermost ones. Typical selection rules are the parallel-outermost selection rule for which all outermost redexes are selected and the leftmost selection rule in which the leftmost outermost redex is selected. The choice of using outermost redexes is motivated by the desire for the evaluation strategy to be lazy.

Definition 5.1.6. A term s is obtained from a term t by a *computation step* using the selection rule S if the following conditions are satisfied.

1. $S(t)$ is a non-empty set, $\{r_i\}$, say.
2. For each i, the redex r_i is α-equivalent to an instance $h_i\theta_i$ of the head of a statement $h_i = b_i$, for some term substitution θ_i.

3. s is the term obtained from t by replacing, for each i, the redex r_i by $b_i\theta_i$.

Each computation step is decidable in the sense that there is an algorithm that can decide for a given subterm whether or not there is an instance of the head of a statement that is α-equivalent to the subterm. This algorithm is similar to the (first-order) unification algorithm. In particular, the undecidability of higher-order unification is not relevant here because α-equivalence is demanded rather than $\beta\eta$-equivalence (or β-equivalence) for higher-order unification.

Definition 5.1.7. A *computation* from a term t is a sequence $\{t_i\}_{i=1}^n$ of terms such that the following conditions are satisfied.

1. $t = t_1$.
2. t_{i+1} is obtained from t_i by a computation step, for $i = 1, \ldots, n-1$.

The term t_1 is called the *goal* of the computation and t_n is called the *answer.*

One interesting aspect of the definition of a computation step is that there are no checks to make sure that the new term obtained from a computation step really is a term, that is, really is type correct. Proposition 5.1.3 below shows that a computation step respects the type requirements. This kind of result is known by the phrase 'run-time type checking is unnecessary'.

Proposition 5.1.2. *Let $h = b$ be a statement and θ an idempotent term substitution such that $h\theta$ is a term. Then $b\theta$ and $h\theta = b\theta$ are terms and $b\theta \precsim h\theta$.*

Proof. This result follows immediately from Proposition 2.5.5 and the facts that $b \precsim h$ and h does not contain repeated free variables. □

Proposition 5.1.3. *Let t be a term and $h = b$ a statement. Suppose there is a subterm r of t at occurrence o and an idempotent term substitution θ such that r is α-equivalent to $h\theta$. Then $t[r/b\theta]_o$ is a term and $t[r/b\theta]_o \precsim t$. Furthermore, $t = t[r/b\theta]_o$ is a term.*

Proof. Since $h = b$ is a statement, $b \precsim h$ and h contains no repeated free variables. Thus, by Proposition 5.1.2, $b\theta$ is a term and $b\theta \precsim h\theta$. Since r is α-equivalent to $h\theta$, $t[r/b\theta]_o$ is a term such that $t[r/b\theta]_o \precsim t$, and $t = t[r/b\theta]_o$ is a term, by Proposition 2.4.6. □

The proof of Proposition 5.1.3 crucially depends on the properties of a statement, in particular, on the requirement that the body of the statement be type-weaker than the head. Here are two examples to show that this requirement cannot be dropped.

Example 5.1.3. Consider the alphabet given by the nullary type constructors M and N, and the constants $p : a \to \Omega$, $q : M \to \Omega$, and $r : N \to \Omega$.

Then the expression $(q\ y) \wedge (r\ y)$ can be obtained by a computation step from the term $(p\ y) \wedge (r\ y)$ and the putative statement $(p\ x) = (q\ x)$ (using the redex $(p\ y)$). However, the expression $(q\ y) \wedge (r\ y)$ is not a term. The problem here is caused by the fact that $(p\ x) = (q\ x)$ is not a statement because $(q\ x)$ is not type-weaker than $(p\ x)$: the x in $(p\ x)$ has relative type a in $(p\ x)$, but the x in $(q\ x)$ has relative type M in $(q\ x)$.

Example 5.1.4. Consider the alphabet given by the nullary type constructors K, M, and N, and the constants $f : a \to K$, $g : K \times K \to \Omega$, $A : M$, and $B : N$.

Then the expression $A = B$ can be obtained by a computation step from the term $(g\ ((f\ A), (f\ B)))$ and the putative statement $(g\ ((f\ x), (f\ y))) = (x = y)$. However, the expression $A = B$ is not a term. The problem here is caused by the fact that $(g\ ((f\ x), (f\ y))) = (x = y)$ is not a statement because $x = y$ is not type-weaker than $(g\ ((f\ x), (f\ y)))$: the x and y have the same relative type a in $x = y$, but they have relative types a and b in $(g\ ((f\ x), (f\ y)))$.

The other condition in a statement that the head does not contain repeated occurrences of free variables is also needed in Proposition 5.1.3.

Example 5.1.5. Let $h = (g\ (x, x))$ and $b = (f\ x)$, where $g : a \times c \to a \times c$ and $f : a \to a \times a$ are constants. Since $(f\ x) \precsim (g\ (x, x))$, $h = b$ would be a statement except for the repeated free variable x in the head. Let $t = (g\ ([], []))$, $r = t$ and $\theta = \{x/[]\}$. Then $h\theta = (g\ ([], []))$ and $b\theta = (f\ [])$. Thus $t[r/b\theta]_\varepsilon = (f\ [])$ is a term, but $t[r/b\theta]_\varepsilon \not\precsim t$, since $(f\ [])$ has type $List\ a \times List\ a$ and $(g\ ([], []))$ has type $List\ a \times List\ c$.

There are two cases of interest in which $h = b$ is not a statement and yet run-time type checking can be avoided.

Proposition 5.1.4. *Let t be a term and $(\lambda x.s\ r)$ a subterm of t at occurrence o. Then $t[(\lambda x.s\ r)/s\{x/r\}]_o$ is a term and $t[(\lambda x.s\ r)/s\{x/r\}]_o \precsim t$. Furthermore, $t = t[(\lambda x.s\ r)/s\{x/r\}]_o$ is a term.*

Proof. The result follows from Part 2 of Proposition 2.6.4 and Proposition 2.4.6. □

In effect, Proposition 5.1.4 shows that the equation $(\lambda x.s\ r) = s\{x/r\}$ which comes from the β-reduction rule can be used as a statement, even though the head is not in the right form.

Proposition 5.1.5. *Let t be a term and $(s = s)$ a subterm of t at occurrence o. Then $t[(s = s)/\top]_o$ is a term and $t[(s = s)/\top]_o \precsim t$. Furthermore, $t = t[(s = s)/\top]_o$ is a term.*

Proof. Since $\top \precsim (s = s)$, the result follows from Proposition 2.4.6. $\square$

In effect, Proposition 5.1.5 shows that the equation $(x = x) = \top$ can be used as a statement, in spite of the repeated variables in the head.

The next proposition establishes a crucial property of computations.

Proposition 5.1.6. *Let P be a program, and s a goal and t an answer of a computation using P. Suppose that I is an interpretation such that, for each statement $h = b$ in P and for each idempotent substitution θ such that $h\theta = b\theta$ is a term, $h\theta = b\theta$ is valid in I. Let φ be a variable assignment with respect to I. Then, for every grounding type substitution η for s, there is a grounding type substitution υ for t such that $\mathcal{V}(s, \eta, I, \varphi) = \mathcal{V}(t, \upsilon, I, \varphi)$.*

Proof. Let $h = b$ be a statement in P and θ an idempotent substitution such that $h\theta = b\theta$ is a term. By Proposition 2.5.5, $b\theta \precsim h\theta$. The result now follows from Proposition 2.8.2 by induction on the length of the computation. $\square$

Note 5.1.1. The importance of Proposition 5.1.6 can be explained as follows. The purpose of carrying out a computation is to discover the meaning (with respect to the intended interpretation, and some variable assignment and grounding type substitution) of some term, say, s. To discover this meaning, the term s is 'simplified' in a computation to a term t, say. According to Proposition 5.1.6, all the meanings of s are contained in the meanings of t. Typically, t is so simple that all its meanings are immediately apparent and thus the meanings of s are as well.

The assumption on the interpretation I in Proposition 5.1.6 is strictly stronger than the assumption that I simply be a model for P. (See Exercise 5.8.) Nevertheless, this assumption on I in Proposition 5.1.6 is one that the intended interpretation of P would surely be expected to satisfy (since it is intended that instances of statements be used in computations).

For later use, it will be convenient to introduce a more flexible notation that extends the syntax of statements. Here is an example to motivate the ideas.

Example 5.1.6. Consider the programming problem of writing some code to implement the subset relation between sets. Here is a possible definition of the function $\subseteq : (a \to \Omega) \to (a \to \Omega) \to \Omega$, which is written infix.

$$\{\} \subseteq s = \top$$
$$\{x \mid x = u\} \subseteq s = u \in s$$
$$\{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} \subseteq s = (\{x \mid \boldsymbol{u}\} \subseteq s) \wedge (\{x \mid \boldsymbol{v}\} \subseteq s).$$

At first sight, all these equations look like terms. However, closer inspection of the third equation reveals that $\boldsymbol{u}$ and $\boldsymbol{v}$ in that equation are not ordinary variables. Intuitively, these are intended to stand for terms (possibly containing x as a free variable). Technically, they are syntactical variables in the meta-language that range over object-level terms. Syntactical variables are distinguished from (ordinary) variables by writing them in bold font.

This use of syntactical variables is common in the presentation of axioms for logics. The third equation in the above example is thus a schema rather than a term in which the syntactical variables $\boldsymbol{u}$ and $\boldsymbol{v}$ range over terms. By replacing the syntactical variables in a schema with suitable terms, a term is obtained from the schema. Syntactical variables are needed at positions in a statement where the usual term substitution mechanism does not give the desired effect because it does not allow free variable capture. For example, if $\boldsymbol{u}$ and $\boldsymbol{v}$ above were (ordinary) variables, it would not be possible to apply a term substitution to the third statement that replaced them by terms containing x as a free variable (and that also retained the bound variable x after the λ).

It is convenient to also adopt a convention that does restrict in some cases the free variables that can appear in the terms replacing syntactical variables. As motivation for this, consider the schema

$$\boldsymbol{u} \wedge (\exists x_1.\cdots\exists x_n.\boldsymbol{v}) = \exists x_1.\cdots\exists x_n.(\boldsymbol{u} \wedge \boldsymbol{v}).$$

This schema is intended to allow the extension of the scope of the existentially quantified variables $x_1, \ldots, x_n$ *provided* $\boldsymbol{u}$ does not contain a free occurrence of any of the x_i, for $i = 1, \ldots, n$. This intention is covered by the following convention.

> If a syntactical variable $\boldsymbol{u}$ occurs both inside the scope of some λx and also outside the scope of all λx s, then $\boldsymbol{u}$ cannot be replaced by an (object-level) term containing x as a free variable.

In general, a statement schema is intended to stand for the collection of statements that can be obtained from the schema by replacing its syntactical variables with terms, applying the preceding convention if necessary. In a few places below, further *ad hoc* restrictions on the syntactical variables will be employed (such as requiring a syntactical variable to range over data constructors or object-level variables only). When using a statement schema in a computation, a choice of terms to replace its syntactical variables is first made. This results in a statement that is then handled as described earlier in this section. Thus a statement schema is a compact way of specifying a (possibly infinite) collection of statements.

5.2 Definitions of Some Basic Functions

This section contains the definitions of $=$, the connectives and quantifiers, and some other basic functions. The emphasis is placed on establishing that these definitions provide the intended meanings of these functions. The next section shows how computation can be performed with the definitions.

The first definition is that of the equality predicate $=$. (Note that a % indicates that the remainder of the line is a comment.)

$$= : a \to a \to \Omega$$

$$(\boldsymbol{C}\ x_1 \ldots x_n = \boldsymbol{C}\ y_1 \ldots y_n) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$

% where $\boldsymbol{C}$ is a data constructor of arity n.

$$(\boldsymbol{C}\ x_1 \ldots x_n = \boldsymbol{D}\ y_1 \ldots y_m) = \bot$$

% where $\boldsymbol{C}$ is a data constructor of arity n,

% $\boldsymbol{D}$ is a data constructor of arity m, and $\boldsymbol{C} \neq \boldsymbol{D}$.

$$((x_1, \ldots, x_n) = (y_1, \ldots, y_n)) = (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$

% where $n = 2, 3, \ldots$.

$$(\lambda x.\boldsymbol{u} = \lambda y.\boldsymbol{v}) = (\mathit{less}\ \lambda x.\boldsymbol{u}\ \lambda y.\boldsymbol{v}) \wedge (\mathit{less}\ \lambda y.\boldsymbol{v}\ \lambda x.\boldsymbol{u})$$

The first two statement schemas in the above definition simply capture the intended meaning of data constructors, while the third captures an important property of tuples. (Note that for the first statement schema, if $n = 0$, then the body is $\top$.)

However, the fourth statement schema is more subtle. In formulations of higher-order logics, it is common for the axioms for equality to include the axiom of extensionality:

$$(f = g) = \forall x.((f\ x) = (g\ x)).$$

However, this axiom is not used here for the reason that it is not computationally useful: showing that $\forall x.((f\ x) = (g\ x))$ is not generally possible as there can be infinitely many values of x to consider. Instead, a special case of the axiom of extensionality is used. Its purpose is to provide a method of checking whether certain abstractions representing finite sets, finite multisets and similar data types are equal. In such cases, one can check for equality in a finite amount of time. The fourth statement schema relies on the two following definitions.

$$\mathit{less} : (a \to b) \to (a \to b) \to \Omega$$

$$\mathit{less}\ \lambda x.\boldsymbol{d}\ z = \top$$

% where $\boldsymbol{d}$ is a default term.

$$\mathit{less}\ (\lambda x.\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w})\ z =$$

$$(\forall x.(\boldsymbol{u} \longrightarrow v = (z\ x))) \wedge (\mathit{less}\ (\mathit{remove}\ \{x \mid \boldsymbol{u}\}\ \lambda x.\boldsymbol{w})\ z)$$

$$\mathit{remove} : (a \to \Omega) \to (a \to b) \to (a \to b)$$

$$\mathit{remove}\ s\ \lambda x.\boldsymbol{d} = \lambda x.\boldsymbol{d}$$

% where $\boldsymbol{d}$ is a default term.

$$\mathit{remove}\ s\ \lambda x.\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w} =$$

$$\lambda x.\mathit{if}\ \boldsymbol{u} \wedge (x \notin s)\ \mathit{then}\ v\ \mathit{else}\ ((\mathit{remove}\ s\ \lambda x.\boldsymbol{w})\ x)$$

The intended meaning of *less* is best given by an illustration. Consider the multisets m and n. Then *less* $m\ n$ is true iff each item in the support of m is also in the support of n and has the same multiplicity there. For sets, *less* is simply the subset relation. If s is a set and m a multiset, then *remove* $s\ m$ returns the multiset obtained from m by removing all the items from its support that are in s. A noteworthy aspect of the definitions of *less* and *remove* is that each consists of two statements, one for the empty abstraction $\lambda x.d$ and one for 'non-empty' abstractions. (Non-empty is quoted since $\{x \mid \mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}\}$ can represent an empty set if both v and $\boldsymbol{w}$ are $\bot$, for example.) There is an analogy here with the definitions of many list-processing functions that have one statement for the empty list and one for non-empty lists.

Using the fourth statement schema for $=$, one can show that the abstractions

$$\lambda x.\mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

and

$$\lambda x.\mathit{if}\ x = 2\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = 1\ \mathit{then}\ \top\ \mathit{else}\ \bot$$

are equal, for example.

The next definition is the obvious one for disequality.

$$\neq\ :\ a \to a \to \Omega$$

$$x \neq y = \neg\,(x = y)$$

The following definitions are for the connectives $\wedge$, $\vee$, and $\neg$.

$$\wedge\ :\ \Omega \to \Omega \to \Omega$$

$$\top \wedge x = x$$
$$x \wedge \top = x$$
$$\bot \wedge x = \bot$$
$$x \wedge \bot = \bot$$
$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z)$$
$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$
$$(\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) \wedge t = \mathit{if}\ u \wedge t\ \mathit{then}\ v\ \mathit{else}\ w \wedge t$$
$$t \wedge (\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) = \mathit{if}\ t \wedge u\ \mathit{then}\ v\ \mathit{else}\ t \wedge w$$
$$\boldsymbol{u} \wedge (\exists x_1.\cdots\exists x_n.\boldsymbol{v}) = \exists x_1.\cdots\exists x_n.(\boldsymbol{u} \wedge \boldsymbol{v})$$
$$(\exists x_1.\cdots\exists x_n.\boldsymbol{u}) \wedge \boldsymbol{v} = \exists x_1.\cdots\exists x_n.(\boldsymbol{u} \wedge \boldsymbol{v})$$
$$\boldsymbol{u} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v} = \boldsymbol{u}\{\boldsymbol{x}/\boldsymbol{t}\} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{t}\}$$

% where $\boldsymbol{x}$ is a variable free in $\boldsymbol{u}$ or $\boldsymbol{v}$ or both, but not free in $\boldsymbol{t}$.

$$\vee : \Omega \to \Omega \to \Omega$$

$$\top \vee x = \top$$
$$x \vee \top = \top$$
$$\bot \vee x = x$$
$$x \vee \bot = x$$
$$(\mathit{if}\ u\ \mathit{then}\ \top\ \mathit{else}\ w) \vee t = \mathit{if}\ u\ \mathit{then}\ \top\ \mathit{else}\ w \vee t$$
$$(\mathit{if}\ u\ \mathit{then}\ \bot\ \mathit{else}\ w) \vee t = (\neg\, u \wedge w) \vee t$$
$$t \vee (\mathit{if}\ u\ \mathit{then}\ \top\ \mathit{else}\ w) = \mathit{if}\ u\ \mathit{then}\ \top\ \mathit{else}\ t \vee w$$
$$t \vee (\mathit{if}\ u\ \mathit{then}\ \bot\ \mathit{else}\ w) = t \vee (\neg\, u \wedge w)$$

$$\neg : \Omega \to \Omega$$

$$\neg \bot = \top$$
$$\neg \top = \bot$$
$$\neg (\neg\, x) = x$$
$$\neg (x \wedge y) = (\neg\, x) \vee (\neg\, y)$$
$$\neg (x \vee y) = (\neg\, x) \wedge (\neg\, y)$$
$$\neg (\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) = \mathit{if}\ u\ \mathit{then}\ \neg\, v\ \mathit{else}\ \neg\, w$$

These definitions are straightforward, except perhaps for the last three statement schemas in the definition of $\wedge$. The second and third last statement schemas allow the scope of existential quantifiers to be extended provided it does not result in free variable capture. (Recall the convention restricting the possible term replacements for syntactical variables.)

The last statement schema allows the elimination of some occurrences of a free variable ($\boldsymbol{x}$, in this case), thus simplifying an expression. A similar statement allowing this kind of simplification also occurs in the definition of Σ below. However, a few words about the expression $\boldsymbol{u} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v}$ are necessary. The intended meaning of this expression is that it is a term such that $(\boldsymbol{x} = \boldsymbol{t})$ is 'embedded conjunctively' inside it. More formally, a term t is embedded conjunctively in t and, if t is embedded conjunctively in r (or s), then t is embedded conjunctively in $r \wedge s$. So, for example, $x = s$ is embedded conjunctively in the term $((p \wedge q) \wedge r) \wedge ((x = s) \wedge (t \wedge u))$.

Next come the definitions of Σ and Π.

$$\Sigma : (a \to \Omega) \to \Omega$$

$$\exists x_1.\cdots\exists x_n.\top = \top$$
$$\exists x_1.\cdots\exists x_n.\bot = \bot$$

$$\exists x_1.\cdots\exists x_n.(\boldsymbol{x} \wedge (x_i = \boldsymbol{u}) \wedge \boldsymbol{y}) =$$
$$\exists x_1.\cdots\exists x_{i-1}.\exists x_{i+1}.\cdots\exists x_n.(\boldsymbol{x}\{x_i/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_i/\boldsymbol{u}\})$$
% where x_i is not free in $\boldsymbol{u}$.
$$\exists x_1.\cdots\exists x_n.(\boldsymbol{u} \vee \boldsymbol{v}) = (\exists x_1.\cdots\exists x_n.\boldsymbol{u}) \vee (\exists x_1.\cdots\exists x_n.\boldsymbol{v})$$
$$\exists x_1.\cdots\exists x_n.(\textit{if } \boldsymbol{u} \textit{ then } \top \textit{ else } \boldsymbol{v}) =$$
$$\textit{if } \exists x_1.\cdots\exists x_n.\boldsymbol{u} \textit{ then } \top \textit{ else } \exists x_1.\cdots\exists x_n.\boldsymbol{v}$$
$$\exists x_1.\cdots\exists x_n.(\textit{if } \boldsymbol{u} \textit{ then } \bot \textit{ else } \boldsymbol{v}) = \exists x_1.\cdots\exists x_n.(\neg\, \boldsymbol{u} \wedge \boldsymbol{v})$$

$$\Pi \ : \ (a \to \Omega) \to \Omega$$

$$\forall x_1.\cdots\forall x_n.(\bot \longrightarrow \boldsymbol{u}) = \top$$
$$\forall x_1.\cdots\forall x_n.(\boldsymbol{x} \wedge (x_i = \boldsymbol{u}) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) =$$
$$\forall x_1.\cdots\forall x_{i-1}.\forall x_{i+1}.\cdots\forall x_n.(\boldsymbol{x}\{x_i/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_i/\boldsymbol{u}\} \longrightarrow \boldsymbol{v}\{x_i/\boldsymbol{u}\})$$
% where x_i is not free in $\boldsymbol{u}$.
$$\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \vee \boldsymbol{v} \longrightarrow \boldsymbol{t}) =$$
$$(\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))$$
$$\forall x_1.\cdots\forall x_n.((\textit{if } \boldsymbol{u} \textit{ then } \top \textit{ else } \boldsymbol{v}) \longrightarrow \boldsymbol{t}) =$$
$$(\forall x_1.\cdots\forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1.\cdots\forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))$$
$$\forall x_1.\cdots\forall x_n.((\textit{if } \boldsymbol{u} \textit{ then } \bot \textit{ else } \boldsymbol{v}) \longrightarrow \boldsymbol{t}) =$$
$$\forall x_1.\cdots\forall x_n.(\neg\, \boldsymbol{u} \wedge \boldsymbol{v} \longrightarrow \boldsymbol{t})$$

For the definition of Σ, recall that $\exists x.t$ stands for $(\Sigma\ \lambda x.t)$. Thus, for example, $\exists x.\top$ stands for $(\Sigma\ \lambda x.\top)$. The definition of Σ is unremarkable except perhaps for the third statement schema that is crucial for supporting the relational style of programming, as will be discussed in the next section. Inspection of the definition for Π indicates that the universal quantifier is always linked with implication, that is, $\longrightarrow$. An example in the next section will illustrate how the definition is used.

Next comes the usual definition for the *if_then_else* function.

$$\textit{if_then_else} \ : \ \Omega \times a \times a \to a$$

$$\textit{if } \top \textit{ then } u \textit{ else } v = u$$
$$\textit{if } \bot \textit{ then } u \textit{ else } v = v$$

Finally, there is the 'definition' corresponding to β-reduction.

$$\lambda x.\boldsymbol{u} : \boldsymbol{\sigma} \to \boldsymbol{\tau}$$

$$\lambda x.\boldsymbol{u}\ t = \boldsymbol{u}\{x/t\}$$
% where $\boldsymbol{\sigma} \to \boldsymbol{\tau}$ is the type of $\lambda x.\boldsymbol{u}$.

5.3 Programming with Abstractions

This section contains a variety of examples to illustrate how computation is performed. Compared with functional programming, the main novelty here is that of programming with abstractions, so the presentation concentrates on this aspect. Programming with abstractions provides a logic programming style of programming in a functional setting.

Relations

Consider the following definitions of the functions *append*, *permute*, *delete*, and *sorted* which have been written in the relational style of logic programming.

$$append : List\ a \times List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} append\ (u, v, w) \ = \ & (u = [] \ \wedge \ v = w) \vee \\ & \exists r.\exists x.\exists y.(u = r \sharp x \ \wedge \ w = r \sharp y \ \wedge \ append\ (x, v, y)) \end{aligned}$$

$$permute : List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} & permute\ ([], x) \ = \ x = [] \\ & permute\ (x \sharp y, w) \ = \\ & \qquad \exists u.\exists v.\exists z.(w = u \sharp v \ \wedge \ delete\ (u, x \sharp y, z) \wedge permute\ (z, v)) \end{aligned}$$

$$delete : a \times List\ a \times List\ a \rightarrow \Omega$$

$$\begin{aligned} & delete\ (x, [], y) \ = \ \bot \\ & delete\ (x, y \sharp z, w) \ = \\ & \qquad (x = y \wedge w = z) \ \vee \ \exists v.(w = y \sharp v \ \wedge \ delete\ (x, z, v)) \end{aligned}$$

$$sorted : List\ Int \rightarrow \Omega$$

$$\begin{aligned} & sorted\ [] \ = \ \top \\ & sorted\ x \sharp y \ = \\ & \qquad if\ y = []\ then\ \top\ else\ \exists u.\exists v.(y = u \sharp v \ \wedge \ x \leq u \ \wedge \ sorted\ y) \end{aligned}$$

The intended meaning of *append* is that it is true iff its third argument is the concatenation of its first two arguments. The intended meaning of *permute* is that it is true iff its second argument is a permutation of its first argument. The intended meaning of *delete* is that it is true iff its third argument is the result of deleting its first argument from its second argument. The intended meaning of *sorted* is that it is true iff its argument is an increasingly ordered

list of integers. As can be seen, the definition of each function has a declarative reading that respects the intended meaning.

The notable feature of the above definitions is the presence of existential quantifiers in the bodies of the statements, so not surprisingly the key statement that makes all this work is concerned with the existential quantifier. To motivate this, consider the computation that results from the goal *append* $(1 \sharp [], 2 \sharp [], x)$. At one point in the computation, the following term is reached:

$$\exists r'.\exists x'.\exists y'.(r' = 1 \ \wedge \ x' = [] \ \wedge \ x = r' \sharp y' \ \wedge \ \mathit{append}\ (x', 2 \sharp [], y').$$

An obviously desirable simplification that can be made to this term is to eliminate the local variable r' since there is a 'value' (that is, 1) for it. This leads to the term

$$\exists x'.\exists y'.(x' = [] \ \wedge \ x = 1 \sharp y' \ \wedge \ \mathit{append}\ (x', 2 \sharp [], y').$$

Similarly, one can eliminate x' to obtain

$$\exists y'.(x = 1 \sharp y' \ \wedge \ \mathit{append}\ ([], 2 \sharp [], y').$$

After some more computation, the answer $x = 1 \sharp 2 \sharp []$ results. Now the statement that makes all this possible is

$$\exists x_1.\cdots\exists x_n.(\boldsymbol{x} \wedge (x_i = \boldsymbol{u}) \wedge \boldsymbol{y}) = \\ \exists x_1.\cdots\exists x_{i-1}.\exists x_{i+1}.\cdots\exists x_n.(\boldsymbol{x}\{x_i/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_i/\boldsymbol{u}\}),$$

which comes from the definition of $\Sigma : (a \to \Omega) \to \Omega$ in Sect. 5.2 and has λ-abstractions in its head.

This example illustrates how the definitions in Sect. 5.2 allow the traditional functional programming style to be extended to encompass the relational style of logic programming. The definitions of predicates look a little different to the way one would write them in, say, Prolog. A mechanical translation of a Prolog definition into one that runs in this style of programming simply involves using the completion of the Prolog definition. The definition here of *append* is essentially the completion of the Prolog version of *append*. Alternatively, one can specialise the completion to the [] and $\sharp$ cases, as has been done here for the definitions of *permute*, *delete*, and *sorted*. One procedural difference of note is that Prolog's method of returning answers one at a time via backtracking is replaced here by returning all answers together as a disjunction (or a set). Thus the goal

$$\mathit{append}\ (x, y, 1 \sharp 2 \sharp [])$$

reduces to the answer

$$(x = [] \wedge y = 1 \sharp 2 \sharp []) \vee (x = 1 \sharp [] \wedge y = 2 \sharp []) \vee (x = 1 \sharp 2 \sharp [] \wedge y = []).$$

Sets

The idea of programming with abstractions can be pushed further to enable programming directly with sets, multisets and other abstractions. First, I deal with sets. Here are the definitions of the functions $\cup$, $\cap$, $\setminus$, and $\subseteq$.

$$\cup : (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$$

$$s \cup t = \{x \mid (x \in s) \vee (x \in t)\}$$

$$\cap : (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$$

$$s \cap t = \{x \mid (x \in s) \wedge (x \in t)\}$$

$$\setminus : (a \to \Omega) \to (a \to \Omega) \to (a \to \Omega)$$

$$s \setminus t = \{x \mid (x \in s) \wedge (x \notin t)\}$$

$$\subseteq : (a \to \Omega) \to (a \to \Omega) \to \Omega$$

$$s \subseteq t = \forall x.(x \in s \longrightarrow x \in t)$$

To illustrate the use of these functions, consider the following definition of the function *likes*.

$$\begin{aligned}
&Mary, Bill, Joe, Fred : Person \\
&Cricket, Football, Tennis : Sport \\
&likes : Person \times Sport \to \Omega \\
&likes = \{(Mary, Cricket), (Mary, Tennis), (Bill, Cricket), \\
&\qquad (Bill, Tennis), (Joe, Tennis), (Joe, Football)\}
\end{aligned}$$

This definition is essentially a database of facts about certain people and the sports they like. Consider first the goal

$$\{Mary, Bill\} \cap \{Joe, Bill\}.$$

Using the statement

$$s \cap t = \{x \mid (x \in s) \wedge (x \in t)\}$$

in the definition of $\cap$, one obtains

$$\{x \mid (x \in \{Mary, Bill\}) \wedge (x \in \{Joe, Bill\})\}$$

and then

$$\{x \mid (\mathit{if}\ x = \mathit{Mary}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \\ \wedge (\mathit{if}\ x = \mathit{Joe}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot)\}$$

by β-reduction. After several uses of the statements

$$(\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) \wedge t = \mathit{if}\ u \wedge t\ \mathit{then}\ v\ \mathit{else}\ w \wedge t$$
$$t \wedge (\mathit{if}\ u\ \mathit{then}\ v\ \mathit{else}\ w) = \mathit{if}\ t \wedge u\ \mathit{then}\ v\ \mathit{else}\ t \wedge w$$
$$\boldsymbol{u} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v} = \boldsymbol{u}\{\boldsymbol{x}/\boldsymbol{t}\} \wedge (\boldsymbol{x} = \boldsymbol{t}) \wedge \boldsymbol{v}\{\boldsymbol{x}/\boldsymbol{t}\}$$

from the definition of $\wedge$, the answer $\{\mathit{Bill}\}$ is obtained. In this example, a novel aspect compared to traditional functional programming is the simplification that has taken place *inside* the body of a λ-abstraction.

For a second example, consider the goal

$$\{x \mid \forall y.(y \in \{\mathit{Cricket}, \mathit{Tennis}\} \longrightarrow \mathit{likes}\ (x, y))\}$$

which reduces via the steps

$$\{x \mid \forall y.((\mathit{if}\ y = \mathit{Cricket}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \\ \longrightarrow \mathit{likes}\ (x, y))\}$$
$$\{x \mid \forall y.((y = \mathit{Cricket}) \longrightarrow \mathit{likes}\ (x, y)) \\ \wedge \forall y.((\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \mathit{likes}\ (x, y))\}$$
$$\{x \mid \mathit{likes}\ (x, \mathit{Cricket}) \wedge \forall y.((\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \mathit{likes}\ (x, y))\}$$

through to

$$\{x \mid (\mathit{if}\ x = \mathit{Mary}\ \mathit{then}\ \top\ \mathit{else}\ \mathit{if}\ x = \mathit{Bill}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \wedge \\ \forall y.((\mathit{if}\ y = \mathit{Tennis}\ \mathit{then}\ \top\ \mathit{else}\ \bot) \longrightarrow \mathit{likes}\ (x, y))\},$$

and so on, to the answer

$$\{\mathit{Mary}, \mathit{Bill}\}.$$

During this computation, use is made of the statements

$$\forall x_1. \cdots \forall x_n.(\boldsymbol{x} \wedge (x_i = \boldsymbol{u}) \wedge \boldsymbol{y} \longrightarrow \boldsymbol{v}) = \\ \forall x_1. \cdots \forall x_{i-1}.\forall x_{i+1}. \cdots \forall x_n.(\boldsymbol{x}\{x_i/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_i/\boldsymbol{u}\} \longrightarrow \boldsymbol{v}\{x_i/\boldsymbol{u}\})$$
$$\forall x_1. \cdots \forall x_n.((\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ \top\ \mathit{else}\ \boldsymbol{v}) \longrightarrow \boldsymbol{t}) = \\ (\forall x_1. \cdots \forall x_n.(\boldsymbol{u} \longrightarrow \boldsymbol{t})) \wedge (\forall x_1. \cdots \forall x_n.(\boldsymbol{v} \longrightarrow \boldsymbol{t}))$$

from the definition of $\Pi : (a \to \Omega) \to \Omega$.

The example in the previous paragraph is reminiscent of list comprehension in functional programming languages, such as Haskell. In fact, one could set the database up as a list of facts and then give Haskell a goal which

would be a list comprehension analogous to the set goal above and obtain a list, say $[Mary, Bill]$, as the answer. Substituting lists for sets in knowledge representation is a standard device to get around the fact that few programming languages support set processing in a sophisticated way. However, sets and lists are actually significantly different types and this shows up, for example, in the different sets of transformations that each type naturally supports. Consequently, I advocate a careful analysis for any particular knowledge representation task to see what types are most appropriate and also that programming languages support a full range of types, including sets and multisets.

Another point to make about the previous example is that it is an illustration of *intensional* set processing. Extensional set processing in which the descriptions of the sets manipulated are explicit representations of the collection of elements in the sets is commonly provided in programming languages. For example, it is straightforward in Haskell to set up an abstract data type for (extensional) sets using lists as the underlying representation. A language such as Java also provides various ways of implementing extensional sets. But the example above is different in that the goal is an intensional representation of a set (in fact, the set $\{Mary, Bill\}$) and the computation is able to reveal this. The ability to process intensional sets and the smooth transition between intensional and extensional set processing are major advantages of the approach to sets advocated here. Similar comments apply to programming with other kinds of abstractions such as multisets.

Consider next the problem of giving a definition for the powerset function that computes the set of all subsets of a given set. Here is the definition.

$$powerset : (a \to \Omega) \to (a \to \Omega) \to \Omega$$

$$powerset\ \{\} = \{\{\}\}$$
$$powerset\ \{x \mid if\ \boldsymbol{u}\ then\ \top\ else\ \boldsymbol{v}\} =$$
$$\{s \mid \exists l.\exists r.((l \in (powerset\ \{x \mid \boldsymbol{u}\})) \wedge (r \in (powerset\ \{x \mid \boldsymbol{v}\})) \wedge (s = l \cup r))\}$$
$$powerset\ \{x \mid if\ \boldsymbol{u}\ then\ \bot\ else\ \boldsymbol{v}\} = powerset\ \{x \mid \neg\, \boldsymbol{u} \wedge \boldsymbol{v}\}$$
$$powerset\ \{x \mid x = t\} = \{\{\}, \{t\}\}$$
$$powerset\ \{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} =$$
$$\{s \mid \exists l.\exists r.((l \in (powerset\ \{x \mid \boldsymbol{u}\})) \wedge (r \in (powerset\ \{x \mid \boldsymbol{v}\})) \wedge (s = l \cup r))\}$$

The first three statements cover the case where the set is represented by a normal abstraction. The fourth and fifth statements are needed if the representation of the set is not a normal abstraction, but has an equality or disjunction at the top level in the body. Of course, if the representation of the set does not match any of the statements, then it will have to be reduced (by using the definitions of other functions) until it does. One can see immediately that each statement in the definition is declaratively correct.

Note the analogy between set processing as illustrated by *powerset* and list processing in which the definition of a list-processing function is broken up into two statements – one for the empty list and one for a non-empty list. In the case of sets, it is convenient to have five cases corresponding to where the body of the set abstraction has the form $\bot$, *if* $\boldsymbol{u}$ *then* $\top$ *else* $\boldsymbol{v}$, *if* $\boldsymbol{u}$ *then* $\bot$ *else* $\boldsymbol{v}$, $x = t$, or $\boldsymbol{u} \vee \boldsymbol{v}$. The fourth and fifth cases arise because of the richness of the set of functions on the booleans. For other kinds of abstractions typically only two cases arise, as is illustrated below for multisets.

As an illustration of the use of *powerset*, the goal

$$powerset\ \{Mary, Bill\}$$

reduces to the answer

$$\{\{\}, \{Mary\}, \{Bill\}, \{Mary, Bill\}\}.$$

The final illustration of set processing is concerned with converting a representation of a set into one that uses a normal abstraction. Consider the function *linearise* with the following definition.

$$linearise : (a \to \Omega) \to (a \to \Omega)$$

$$linearise\ \{\} = \{\}$$
$$linearise\ \{x \mid if\ \boldsymbol{u}\ then\ \top\ else\ \boldsymbol{v}\} = (linearise\ \{x \mid \boldsymbol{u}\}) \cup (linearise\ \{x \mid \boldsymbol{v}\})$$
$$linearise\ \{x \mid if\ \boldsymbol{u}\ then\ \bot\ else\ \boldsymbol{v}\} = linearise\ \{x \mid \neg\, \boldsymbol{u} \wedge \boldsymbol{v}\}$$
$$linearise\ \{x \mid x = t\} = \{x \mid if\ x = t\ then\ \top\ else\ \bot\}$$
$$linearise\ \{x \mid \boldsymbol{u} \vee \boldsymbol{v}\} = (linearise\ \{x \mid \boldsymbol{u}\}) \cup (linearise\ \{x \mid \boldsymbol{v}\})$$

Because of the rich set of functions on the booleans, there are many different ways of representing a finite set. For example, the set $\{1, 2\}$ can be represented by the term

$$\{x \mid (x = 1) \vee (x = 2)\},$$

as well as by the normal abstraction

$$\lambda x.if\ x = 1\ then\ \top\ else\ if\ x = 2\ then\ \top\ else\ \bot.$$

The function *linearise* provides a way to turn the former into the latter, by means of the following computation.

$$linearise\ \{x \mid (x = 1) \vee (x = 2)\}$$
$$linearise\ \{x \mid x = 1\} \cup linearise\{x \mid x = 2\}$$
$$\{x \mid if\ x = 1\ then\ \top\ else\ \bot\} \cup \{x \mid if\ x = 2\ then\ \top\ else\ \bot\}$$
$$\{y \mid (y \in \{x \mid if\ x{=}1\ then\ \top\ else\ \bot\}) \vee (y \in \{x \mid if\ x{=}2\ then\ \top\ else\ \bot\})\}$$
$$\{y \mid (if\ y = 1\ then\ \top\ else\ \bot) \vee (if\ y = 2\ then\ \top\ else\ \bot)\}$$
$$\{y \mid if\ y = 1\ then\ \top\ else\ (\bot \vee (if\ y = 2\ then\ \top\ else\ \bot))\}$$
$$\{y \mid if\ y = 1\ then\ \top\ else\ if\ y = 2\ then\ \top\ else\ \bot\}.$$

Multisets

The next group of definitions provide basic multiset processing.

$$\uplus : (a \rightarrow Nat) \rightarrow (a \rightarrow Nat) \rightarrow (a \rightarrow Nat)$$

$$\lambda x.0 \uplus m = m$$
$$(\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \uplus m =$$
$$\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v + (m\ t)\ \mathit{else}\ (\lambda x.\boldsymbol{w} \uplus (\mathit{remove}\ \{t\}\ m))\ x$$

$$\boxminus : (a \rightarrow Nat) \rightarrow (a \rightarrow Nat) \rightarrow (a \rightarrow Nat)$$

$$\lambda x.0 \boxminus m = \lambda x.0$$
$$(\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \boxminus m =$$
$$\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v \dot{-} (m\ t)\ \mathit{else}\ (\lambda x.\boldsymbol{w} \boxminus m)\ x$$

$$\sqcup : (a \rightarrow Nat) \rightarrow (a \rightarrow Nat) \rightarrow (a \rightarrow Nat)$$

$$\lambda x.0 \sqcup m = m$$
$$(\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \sqcup m =$$
$$\lambda x.\mathit{if}\ x = t\ \mathit{then}\ \mathit{max}\ v\ (m\ t)\ \mathit{else}\ (\lambda x.\boldsymbol{w} \sqcup (\mathit{remove}\ \{t\}\ m))\ x$$

$$\sqcap : (a \rightarrow Nat) \rightarrow (a \rightarrow Nat) \rightarrow (a \rightarrow Nat)$$

$$\lambda x.0 \sqcap m = \lambda x.0$$
$$(\lambda x.\mathit{if}\ x = t\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \sqcap m =$$
$$\lambda x.\mathit{if}\ x = t\ \mathit{then}\ \mathit{min}\ v\ (m\ t)\ \mathit{else}\ (\lambda x.\boldsymbol{w} \sqcap m)\ x$$

$$\sqsubseteq : (a \rightarrow Nat) \rightarrow (a \rightarrow Nat) \rightarrow \Omega$$

$$\lambda x.0 \sqsubseteq m = \top$$
$$(\lambda x.\mathit{if}\ \boldsymbol{u}\ \mathit{then}\ v\ \mathit{else}\ \boldsymbol{w}) \sqsubseteq m =$$
$$(\forall x.(\boldsymbol{u} \longrightarrow v \leq (m\ x))) \wedge ((\mathit{remove}\ \{x \mid \boldsymbol{u}\}\ \lambda x.\boldsymbol{w}) \sqsubseteq m)$$

$$\sqsubset\!\!\!- : a \rightarrow (a \rightarrow Nat) \rightarrow \Omega$$

$$x \sqsubset\!\!\!- m = (m\ x) > 0$$

The function $\uplus$ computes the multiset union, the function $\boxminus$ computes the multiset difference, the function $\sqcup$ computes the pairwise maximum, and the function $\sqcap$ computes the pairwise minimum of two multisets. (Note that $\dot{-}$ is

the monus function, which is similar to subtraction except a negative result is rounded up to 0.) The function $\sqsubseteq$ is multiset inclusion and the function $\sqsubseteq$ is multiset membership. As an illustration of the use of $\sqcap$, the goal

$$(\lambda x.if\ x = A\ then\ 42\ else\ if\ x = B\ then\ 21\ else\ 0) \\ \sqcap\ (\lambda x.if\ x = A\ then\ 16\ else\ if\ x = C\ then\ 4\ else\ 0)$$

reduces to the answer

$$\lambda x.if\ x = A\ then\ 16\ else\ 0.$$

A Computation for a Learning Problem

To tie the above ideas together in the learning context of this book, consider the evaluation of the term

$$setExists_1\ (\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium))) \\ \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}.$$

The computation, which uses the leftmost selection rule and has the redexes underlined, is as follows.

$$\underline{setExists_1\ (\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium)))} \\ \underline{\{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}}$$

$$\exists x.(\underline{((\wedge_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium)))\ x)} \wedge \\ (x \in \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}))$$

$$\exists x.((\underline{((projMake \circ (= Abloy))\ x)} \wedge ((projLength \circ (= Medium))\ x)) \wedge \\ (x \in \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}))$$

$$\exists x.((((= Abloy)\ (projMake\ x)) \wedge \underline{((projLength \circ (= Medium))\ x)}) \wedge \\ (x \in \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}))$$

$$\exists x.((((= Abloy)\ (projMake\ x)) \wedge ((= Medium)\ (projLength\ x))) \wedge \\ \underline{(x \in \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\})})$$

$$\exists x.\underline{((((= Abloy)\ (projMake\ x)) \wedge ((= Medium)\ (projLength\ x))) \wedge} \\ \underline{(if\ x = (Abloy, 3, Short, Normal)\ then\ \top\ else} \\ \underline{if\ x = (Abloy, 4, Medium, Broad)\ then\ \top\ else\ \bot))}$$

$$\begin{array}{r}
\underline{\exists x.\mathit{if}\ ((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge} \\
\underline{(x = (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}))) \ \mathit{then}\ \top\ \mathit{else}} \\
\underline{((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge} \\
\underline{(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))}
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ \underline{\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge} \\
\underline{(x = (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal})))}\ \mathit{then}\ \top\ \mathit{else} \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ (((=\mathit{Abloy})\ \underline{(\mathit{projMake}\ (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}))}) \wedge \qquad\qquad\qquad\qquad \\
((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal})))) \ \mathit{then}\ \top\ \mathit{else} \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ (\underline{((=\mathit{Abloy})\ \mathit{Abloy})} \wedge \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal})))) \ \mathit{then}\ \top\ \mathit{else} \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ \underline{(\top \wedge} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
\underline{((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}))))}\ \mathit{then}\ \top\ \mathit{else} \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ ((=\mathit{Medium})\ \underline{(\mathit{projLength}\ (\mathit{Abloy}, 3, \mathit{Short}, \mathit{Normal}))})\ \mathit{then}\ \top\ \mathit{else} \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\begin{array}{r}
\mathit{if}\ \underline{((=\mathit{Medium})\ \mathit{Short})}\ \mathit{then}\ \top\ \mathit{else} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \\
\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \\
(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))
\end{array}$$

$$\underline{\mathit{if}\ \bot\ \mathit{then}\ \top\ \mathit{else}}$$
$$\underline{\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge}$$
$$\underline{(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))}$$

$$\exists x.\underline{((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge}$$
$$\underline{(\mathit{if}\ x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})\ \mathit{then}\ \top\ \mathit{else}\ \bot))}$$

$$\underline{\exists x.\mathit{if}\ ((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge}$$
$$\underline{(x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})))\ \mathit{then}\ \top\ \mathit{else}}$$
$$\underline{((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)}$$

$$\mathit{if}\ \underline{\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge}$$
$$\underline{(x = (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})))}\ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\mathit{if}\ (((=\mathit{Abloy})\ \underline{(\mathit{projMake}\ (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}))}) \wedge$$
$$((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})))) \ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\mathit{if}\ (\underline{((=\mathit{Abloy})\ \mathit{Abloy})} \wedge$$
$$((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad})))) \ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\mathit{if}\ \underline{(\top \wedge}$$
$$\underline{((=\mathit{Medium})\ (\mathit{projLength}\ (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}))))} \ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\mathit{if}\ ((=\mathit{Medium})\ \underline{(\mathit{projLength}\ (\mathit{Abloy}, 4, \mathit{Medium}, \mathit{Broad}))})\ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\mathit{if}\ \underline{((=\mathit{Medium})\ \mathit{Medium})}\ \mathit{then}\ \top\ \mathit{else}$$
$$\exists x.((((=\mathit{Abloy})\ (\mathit{projMake}\ x)) \wedge ((=\mathit{Medium})\ (\mathit{projLength}\ x))) \wedge \bot)$$

$$\frac{\frac{\textit{if } \top \textit{ then } \top \textit{ else}}{\exists x.((((= \textit{Abloy})\ (\textit{projMake } x)) \wedge ((= \textit{Medium})\ (\textit{projLength } x))) \wedge \bot)}}{\top.}$$

Bibliographical Notes

The particular approach to computation in this chapter starts from a traditional functional programming computational model and extends it so that logic programming idioms can also be supported. Thus the resulting programming language, called Escher [53, 54], is technically a functional logic programming language [31]. There have been many previous efforts to design functional logic programming languages some of which are reported in [31]. An early influential collection of papers on this topic is in [17]. The most important of the current functional logic languages but based on a different computational model is Curry [32], which has attracted an international design and implementation effort.

An extensive and detailed study of the *efficient* implementation of the computational model of this chapter in the context of the Escher language was undertaken by Eder [21, 22, 23, 24]. Eder designed an abstract machine for Escher based on an extension of a functional abstract machine. Reasonable performance was obtained, but further optimisations of this approach will be needed before it becomes truly competitive.

The particular formulation of the definitions of the connectives and quantifiers in Sect. 5.2 and the programming with abstractions paradigm in Sect. 5.3 first appeared in [54, 55, 56]. Another approach to providing sets and multisets in declarative programming languages is given in [19].

A 'run-time type checking is unnecessary' result in a functional programming context appeared in [60]. A first-order result of this kind for Prolog was first given in [67] and later in [35].

The definition of the completion of a logic program is due to Clark [12]. See also [52].

Exercises

5.1 Consider the following definition of the function *member*.

$$\textit{member} : a \to \textit{List } a \to \Omega$$

$$\textit{member } x\ [\,] = \bot$$
$$\textit{member } x\ y \sharp z = (x = y) \vee \textit{member } x\ z.$$

(i) Evaluate *member* 2 [1, 2, 3].
(ii) Evaluate *member* 4 [1, 2, 3].
(iii) Evaluate *member* x [1, 2, 3].

5.2 For the function *append* of Sect. 5.3, show that the goal

$$append\ (x, y, 1 \sharp [])$$

reduces to the answer

$$(x = [] \wedge y = 1 \sharp []) \vee (x = 1 \sharp [] \wedge y = []).$$

5.3 For the function *powerset* of Sect. 5.3, show that the goal

$$powerset\ \{Mary, Bill\}$$

reduces to the answer

$$\{\{\}, \{Mary\}, \{Bill\}, \{Mary, Bill\}\}.$$

5.4 Write a program using the programming with abstractions paradigm that solves the following problem: find the smallest 6 digit number consisting of two 1s separated by 1 digit, two 2s separated by two digits, and two 3s separated by three digits.
[Hint: Represent numbers by the list of their digits. Write a predicate *generate* on lists of integers that generates all candidate numbers that satisfy the conditions. Then write a function *smallest* that picks the smallest list of integers out of the set of such lists (where 'smallest' is according to the order on the numbers represented by the lists of integers). The required number is then given by *smallest* $\{x \mid generate\ x\}$.]

5.5 Give the definition of a function $card : (a \rightarrow \Omega) \rightarrow Nat$ that computes the cardinality of a finite set.

5.6 Let

$$ifsome\ \exists x_1. \cdots \exists x_n.b\ then\ c\ else\ d$$

be syntactic sugar for

$$(\exists x_1. \cdots \exists x_n.(b \wedge c)) \ \vee \ (\neg \exists x_1. \cdots \exists x_n.b \wedge d),$$

and consider the following statements

$$ifsome\ \exists x_1. \cdots \exists x_n.\top\ then\ \boldsymbol{x}\ else\ y = \exists x_1. \cdots \exists x_n.\boldsymbol{x}$$

$$ifsome\ \exists x_1. \cdots \exists x_n.\bot\ then\ \boldsymbol{x}\ else\ y = y$$

$$ifsome\ \exists x_1. \cdots \exists x_n.(\boldsymbol{x} \wedge (x_i = \boldsymbol{u}) \wedge \boldsymbol{y})\ then\ \boldsymbol{z}\ else\ v = \\ ifsome\ \exists x_1. \cdots \exists x_{i-1}.\exists x_{i+1}. \cdots \exists x_n.(\boldsymbol{x}\{x_i/\boldsymbol{u}\} \wedge \boldsymbol{y}\{x_i/\boldsymbol{u}\}) \\ then\ \boldsymbol{z}\{x_i/\boldsymbol{u}\}\ else\ v$$

% where x_i is not free in $\boldsymbol{u}$.

$$ifsome\ \exists x_1. \cdots \exists x_n.(\boldsymbol{x} \vee (x_i = \boldsymbol{u}) \vee \boldsymbol{y})\ then\ \boldsymbol{z}\ else\ v = \\ \exists x_1. \cdots \exists x_n.((\boldsymbol{x} \vee (x_i = \boldsymbol{u}) \vee \boldsymbol{y}) \wedge \boldsymbol{z}).$$

(i) What is the meaning of the function *lookup* having the following definition?

$$lookup : Int \times Person \times List\ (Int \times Person) \times List\ (Int \times Person) \to \Omega$$

$$\begin{aligned}&lookup\ (key, value, assoc_list, new_assoc_list) =\\&\quad ifsome\ \exists v.member\ (key, v)\ assoc_list\\&\quad then\ value = v \land new_assoc_list = assoc_list\\&\quad else\ new_assoc_list = (key, value) \sharp\ assoc_list.\end{aligned}$$

(ii) By constructing the appropriate computation, show that the term

$$lookup\ (5, value, [(4, Fred), (5, Mary)], list)$$

reduces to

$$(value = Mary) \land (list = [(4, Fred), (5, Mary)]).$$

(iii) By constructing the appropriate computation, show that the term

$$lookup\ (5, value, [(4, Fred), (5, Mary), (5, Bill)], list)$$

reduces to

$$\begin{aligned}&((value = Mary) \land (list = [(4, Fred), (5, Mary), (5, Bill)])) \lor\\&((value = Bill) \land (list = [(4, Fred), (5, Mary), (5, Bill)])).\end{aligned}$$

(iv) By constructing the appropriate computation, show that the term

$$lookup\ (6, Joe, [(4, Fred), (5, Mary), (5, Bill)], list)$$

reduces to

$$list = [(6, Joe), (4, Fred), (5, Mary), (5, Bill)].$$

5.7 Redo the evaluation of

$$\begin{aligned}&setExists_1\ (\land_2\ (projMake \circ (= Abloy))\ (projLength \circ (= Medium)))\\&\qquad\qquad \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}\end{aligned}$$

using instead the representation

$$\{x \mid (x = (Abloy, 3, Short, Normal)) \lor (x = (Abloy, 4, Medium, Broad))\}$$

for the set $\{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad)\}$. Can you draw any conclusions about the relative merits of this kind of representation for sets compared with normal abstractions?

5.8 Give an example of an interpretation I, a statement $h = b$, and an idempotent substitution θ such that $h\theta = b\theta$ is a term and $h = b$ is valid in I, but $h\theta = b\theta$ is not valid in I.

5.9 Prove or disprove the following conjecture.
Let P be a program. Then, for every computation with goal s and answer t, $s = t$ is a logical consequence of P.

5.10 (Open problem) Prove or disprove the following conjecture.
Let P be a program, and s a goal and t an answer of a computation. Let I be a model for P and φ a variable assignment with respect to I. Then, for every grounding type substitution η for s, there is a grounding type substitution υ for t such that $\mathcal{V}(s, \eta, I, \varphi) = \mathcal{V}(t, \upsilon, I, \varphi)$.

5.11 (Open problem) Give a definition of the concept of a schema by extending the definition of a term in Chap. 2 to include syntactical variables. Also extend the concept of type weaker for terms in Chap. 2 to type weaker for schemas. Hence give a suitable definition of the concept of statement schema (that covers its usage in this chapter) and establish a 'run-time type checking is unnecessary' result for statement schemas.

5.12 (Open problem) Find an implementation strategy for the programming with abstractions paradigm which has the properties that (i) functional programs that do not use this paradigm (for example, Haskell programs) run with the same efficiency on the implementation as they do on a standard functional implementation, and (ii) programs that correspond directly to Prolog programs run with the same efficiency on the implementation as the corresponding Prolog programs do on a Prolog system.

6. Learning

This chapter discusses a widely applicable learning paradigm that leads to comprehensible theories, provides an overview of the ALKEMY learning system, and also gives some illustrations of this approach to learning.

6.1 Decision-Tree Learning

Decision-tree learning systems have a long history of development and successful application. These systems are based on the following intuitively appealing idea. To build a classifier from a set of examples, find a criterion that partitions the examples into two sets which are purer in the distribution of classes that they contain than the original set; apply this process recursively on the child nodes until the leaf nodes of the tree are sufficiently pure; and then use the resulting decision tree as the induced classifier. (The formulation here focuses on *binary* decision trees. More generally, one can partition into more than two sets.) Decision-tree learning is one of few learning methods that provides comprehensible hypotheses and this explains the concentration on that approach here.

An important issue for decision-tree algorithms is to decide under what situations a node should be split and, if so, what predicate should be used to split it. Given that the overall aim of the algorithm is to produce a tree with high predictive accuracy, it is natural to propose to use accuracy as the criterion for node splitting. (The accuracy of a set of examples is the fraction of the examples in the majority class of the set; the precise definition is given below.) A variety of other heuristics mostly based on entropy could also be used. Accuracy has the advantage of providing a basis for a safe pruning method as shall be indicated below. I now give the theoretical background for the accuracy heuristic.

First, some definitions are presented. Suppose there are c classes in all. Let $\mathcal{E}$ be a (non-empty) set of examples, N the number of examples in $\mathcal{E}$, n_i the number of examples in $\mathcal{E}$ in the ith class, and $p_i = n_i/N$, for $i = 1, \ldots, c$.

Definition 6.1.1. The *majority class* of $\mathcal{E}$ is defined to be the class to which the greatest number of examples in $\mathcal{E}$ belong. (If there are two or more classes

J. W. Lloyd, *Logic for Learning*
© J. W. Lloyd 2003

with the same maximal number of examples, one is chosen arbitrarily as the majority class.)

Definition 6.1.2. The *accuracy*, $A_{\mathcal{E}}$, of the set $\mathcal{E}$ of examples is defined by

$$A_{\mathcal{E}} \equiv p_M,$$

where M is the index of the majority class of $\mathcal{E}$.

The accuracy is the fraction of examples which are correctly classified on the basis that the majority class gives the classification. $A_{\mathcal{E}}$ lies in the range $[1/c, 1]$, where larger values correspond intuitively to purer sets of examples. If all the examples come from the same class, then $A_{\mathcal{E}} = 1$. If the examples are evenly distributed amongst the classes, then $A_{\mathcal{E}} = 1/c$. I also define the accuracy of a partition of a set of examples.

Definition 6.1.3. Let $\mathcal{P} = \{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ be a partition of a set of N examples, where there are N_j examples in $\mathcal{E}_j$, for $j = 1, \ldots, m$. Then the *accuracy*, $A_{\mathcal{P}}$, of the partition $\mathcal{P}$ is defined by

$$A_{\mathcal{P}} \equiv \sum_{j=1}^{m} \frac{N_j}{N} A_{\mathcal{E}_j}.$$

Thus $A_{\mathcal{P}}$ is the weighted average of the accuracies of the individual sets of examples in the partition. $A_{\mathcal{P}}$ lies in the range $[1/c, 1]$, where larger values correspond to partitions that are more accurate.

The decision-tree learning algorithm used by ALKEMY makes binary splits at each node in the tree, so I now develop some material about binary partitions that will be needed. For this, it will be convenient to define a binary partition of a set of examples to be an *ordered* pair.

Definition 6.1.4. A *binary partition* of a set of examples $\mathcal{E}$ is an ordered pair $(\mathcal{E}_1, \mathcal{E}_2)$ such that $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$ and $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$.

If $\mathcal{E}$ is a set of examples, a predicate p induces a binary partition $(\mathcal{E}_1, \mathcal{E}_2)$ of $\mathcal{E}$, where $\mathcal{E}_1$ is the set of examples such that the predicate evaluates to true on the individual in the example and $\mathcal{E}_2$ is the set of examples such that the predicate evaluates to false on the individual in the example. Now, given a predicate p and a rewrite, predicate construction creates a new predicate p' such that $p \Leftarrow p'$, assuming the predicate rewrite system is monotone (see Proposition 4.5.3). Thus the binary partition $(\mathcal{E}'_1, \mathcal{E}'_2)$ of $\mathcal{E}$ induced by p' has the property that $\mathcal{E}'_1 \subseteq \mathcal{E}_1$. These considerations lead to the following definition.

Definition 6.1.5. Let $\mathcal{E}$ be a set of examples and $(\mathcal{E}_1, \mathcal{E}_2)$ a binary partition of $\mathcal{E}$. Then a binary partition $(\mathcal{E}'_1, \mathcal{E}'_2)$ of $\mathcal{E}$ is said to be a *refinement* of $(\mathcal{E}_1, \mathcal{E}_2)$ if $\mathcal{E}'_1 \subseteq \mathcal{E}_1$.

Next a measure for binary partitions is introduced that allows safe pruning of the search space of predicates.

Definition 6.1.6. Let $\mathcal{P} = (\mathcal{E}_1, \mathcal{E}_2)$ be a binary partition of a set $\mathcal{E}$ of N examples, where n_i is the number of examples in $\mathcal{E}$ in the ith class and $n_{j,i}$ is the number of examples in $\mathcal{E}_j$ in the ith class, for $j = 1, 2$ and $i = 1, \ldots, c$. Then the *refinement bound*, $B_\mathcal{P}$, of the binary partition $\mathcal{P}$ is defined by

$$B_\mathcal{P} \equiv (\max_i \{n_i + \max_{k \neq i} n_{1,k}\})/N.$$

I will show below that $B_\mathcal{P}$ is an upper bound for the accuracy of any binary partition that is a refinement of $\mathcal{P}$ and there exists a refinement of $\mathcal{P}$ that has accuracy $B_\mathcal{P}$. The intuitive idea behind the definition of $B_\mathcal{P}$ is that the refinement of $\mathcal{P}$ having the greatest accuracy can be obtained by moving all examples in one class, say i_0, from $\mathcal{E}_1$ across to $\mathcal{E}_2$ making i_0 the majority class in (the new) $\mathcal{E}_2$ and leaving $\text{argmax}_{k \neq i_0} n_{1,k}$ as the majority class in (the new) $\mathcal{E}_1$. Here is an example to illustrate the concept of refinement bound. In the following, I denote by $(n_1, \ldots, n_c)$ a set of examples with n_i examples from the ith class, for $i = 1, \ldots, c$.

Example 6.1.1. Let $\mathcal{E} = (6, 9, 3, 2)$ and suppose $\mathcal{P} = ((2, 1, 0, 2), (4, 8, 3, 0))$. Then $A_\mathcal{P} = 10/20$ and $B_\mathcal{P} = 11/20$. If $\mathcal{Q} = ((0, 9, 0, 0), (6, 0, 3, 2))$, then $A_\mathcal{Q} = B_\mathcal{Q} = 15/20$.

The following proposition collects the basic properties of refinement bounds.

Proposition 6.1.1. *Let $\mathcal{E}$ be a set of examples and $\mathcal{P}$ a binary partition of $\mathcal{E}$.*

1. *If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_\mathcal{P}$. In particular, $A_\mathcal{P} \leq B_\mathcal{P}$.*
2. *If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $B_{\mathcal{P}'} \leq B_\mathcal{P}$.*
3. *There is a refinement $\mathcal{R}$ of $\mathcal{P}$ such that $A_\mathcal{R} = B_\mathcal{P}$.*
4. *$A_\mathcal{P} = B_\mathcal{P}$ iff, for all refinements $\mathcal{P}'$ of $\mathcal{P}$, $A_{\mathcal{P}'} \leq A_\mathcal{P}$.*

Proof. 1. Let $\mathcal{P}$ be $(\mathcal{E}_1, \mathcal{E}_2)$ and $\mathcal{P}'$ be $(\mathcal{E}_1', \mathcal{E}_2')$, where $\mathcal{E}_1' \subseteq \mathcal{E}_1$. Let $n_{1,i}$ be the number of examples in $\mathcal{E}_1$ in the ith class and M_2' the index of the majority class of $\mathcal{E}_2'$. Consider the binary partition $\mathcal{Q}$ obtained from $\mathcal{P}'$ by moving across any remaining examples from class M_2' in $\mathcal{E}_1'$ to $\mathcal{E}_2'$. Clearly $A_{\mathcal{P}'} \leq A_\mathcal{Q}$, since the examples so moved will be correctly classified after the move. But $A_\mathcal{Q} \leq B_\mathcal{P}$, since $A_\mathcal{Q} \leq (\{n_{M_2'} + \max_{k \neq M_2'} n_{1,k}\})/N$ and $B_\mathcal{P} = (\max_i \{n_i + \max_{k \neq i} n_{1,k}\})/N$. Hence the result.

2. Let $n_{1,i}'$ be the number of examples in $\mathcal{E}_1'$ in the ith class, for $i = 1, \ldots, c$. Clearly $\max_{k \neq i} n_{1,k}' \leq \max_{k \neq i} n_{1,k}$, for $i = 1, \ldots, c$. Hence $B_{\mathcal{P}'} \leq B_\mathcal{P}$.

3. Let i_0 be an index for which the maximum in the definition of $B_\mathcal{P}$ is achieved. Construct the binary partition $\mathcal{R} = (\mathcal{E}_1', \mathcal{E}_2')$ from $\mathcal{P}$ by moving all examples in class i_0 from $\mathcal{E}_1$ across to $\mathcal{E}_2$. I claim that the majority class of $\mathcal{E}_2'$ is i_0. Suppose not. Then there is another class i_1, say,

such that $\mathcal{E}_2'$ contains strictly more examples of class i_1 than class i_0. There are two cases to consider. Suppose that $\text{argmax}_{k \neq i_0}\ n_{1,k} \neq i_1$. Thus $n_{i_1} + \max_{k \neq i_1}\ n_{1,k} > n_{i_0} + \max_{k \neq i_0}\ n_{1,k}$ and there is a contradiction to the definition of i_0. Otherwise, suppose that $\text{argmax}_{k \neq i_0}\ n_{1,k} = i_1$. Thus $n_{i_1} > n_{i_0} + \max_{k \neq i_0}\ n_{1,k}$ and there is a contradiction again. Thus i_0 is the majority class in $\mathcal{E}_2'$ and it follows that $A_{\mathcal{R}} = B_{\mathcal{P}}$.

4. Suppose that $A_{\mathcal{P}} = B_{\mathcal{P}}$. If $\mathcal{P}'$ is a refinement of $\mathcal{P}$, then $A_{\mathcal{P}'} \leq B_{\mathcal{P}}$, by Part 1. Hence $A_{\mathcal{P}'} \leq A_{\mathcal{P}}$. Conversely, suppose $A_{\mathcal{P}'} \leq A_{\mathcal{P}}$, for all refinements $\mathcal{P}'$ of $\mathcal{P}$. By Part 3, there is a refinement $\mathcal{R}$ of $\mathcal{P}$ such that $A_{\mathcal{R}} = B_{\mathcal{P}}$. Hence $A_{\mathcal{P}} \leq B_{\mathcal{P}} = A_{\mathcal{R}} \leq A_{\mathcal{P}}$ and the result follows. □

Part 1 of Proposition 6.1.1 is used by the ALKEMY learning system to prune the search space when searching for a predicate to split a node. During this search, the system records the best partition $\mathcal{P}$ found so far and its associated accuracy $A_{\mathcal{P}}$. When investigating a new partition $\mathcal{Q}$, the quantity $B_{\mathcal{Q}}$ is calculated. According to the proposition, if $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, then the partition $\mathcal{Q}$ and all its refinements can be safely pruned. The effect of this pruning is often to greatly reduce the part of the search space that has to be searched. Note carefully that the pruning is in the *search space of predicates* for splitting a particular node *not* in the decision tree itself. Here is an example to illustrate how this works.

Example 6.1.2. Let the set of examples be $(6, 9, 3, 2)$. Suppose the best partition found so far is $\mathcal{P} = ((5,3,0,2),(1,6,3,0))$, which has accuracy $11/20$. Suppose that later on in the search the partition $\mathcal{Q} = ((2,4,0,1),(4,5,3,1))$ is being investigated. Note that $B_{\mathcal{Q}} = 11/20$. Since $B_{\mathcal{Q}} \leq A_{\mathcal{P}}$, the proposition shows that $\mathcal{Q}$ and its refinements can be pruned.

On the other hand, consider the partition $\mathcal{R} = ((6,5,3,2),(0,4,0,0))$, for which $B_{\mathcal{R}} = 15/20$. Thus $\mathcal{R}$ has refinements, which could be found by the system, whose accuracies exceed that of $\mathcal{P}$. (The refinements actually investigated depend upon the hypothesis language, of course.) Thus $\mathcal{R}$ should not be pruned.

I now turn to a detailed description of the decision-tree algorithm used by ALKEMY. Figure 6.1 gives an overview of this algorithm. The input is a set of examples, a predicate rewrite system, and a parameter to be explained below. Each example has the form (t, C), where t is a basic term representing an individual and C is its class. The algorithm builds a decision tree, labels each leaf node of the tree by its majority class, and then performs post-pruning on this tree to produce the final decision tree. The post-pruning algorithm used is called error-complexity post-pruning; other techniques could also be employed.

Figure 6.2 contains a description of the algorithm that builds a decision-tree. It calls upon another algorithm in Fig. 6.3 to provide suitable predicates to perform the splitting. A node is only split if a predicate can be found such that the accuracy of the partition induced by the predicate is strictly greater

function *Learn*$(\mathcal{E}, \rightarrowtail, P)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, *prune* parameter;

tree := *BuildTree*$(\mathcal{E}, \rightarrowtail, P)$
label each leaf node of *tree* by its majority class;
tree := *Postprune*(*tree*);
return *tree*;

Fig. 6.1. Decision-tree learning algorithm

than the accuracy of the set of examples at the node. In Fig. 6.2, *tree.right* is the right subtree of *tree*; similarly, for *tree.left*. It is understood that the branch from the node to the left subtree is labelled by p and the branch to the right subtree is labelled by $\neg p$; this information is used to extract the definition of the induced function from the decision tree.

function *BuildTree*$(\mathcal{E}, \rightarrowtail, P)$ **returns** a decision tree;
inputs: $\mathcal{E}$, a set of examples;
$\rightarrowtail$, a predicate rewrite system;
P, *prune* parameter;

tree := single node (with examples $\mathcal{E}$);
$p :=$ *Predicate*$(\mathcal{E}, \rightarrowtail, P)$;
$\mathcal{P} :=$ binary partition induced by p;
if $A_{\mathcal{P}} \leq A_{\mathcal{E}}$ **then return** *tree*;
$\mathcal{E}_1 := \{(t, C) \mid (t, C) \in \mathcal{E}$ and $(p\ t)$ evaluates to $\top\}$;
$\mathcal{E}_2 := \{(t, C) \mid (t, C) \in \mathcal{E}$ and $(p\ t)$ evaluates to $\bot\}$;
tree.left := *BuildTree*$(\mathcal{E}_1, \rightarrowtail, P)$;
tree.right := *BuildTree*$(\mathcal{E}_2, \rightarrowtail, P)$;
return *tree*;

Fig. 6.2. Tree building algorithm

The most interesting aspect of the decision-tree algorithm is the way it searches the space of predicates to find a good predicate to split a node, as shown in Fig. 6.3. The algorithm there is based on the predicate enumeration process described in Sects. 4.4 and 4.6. The function *Predicate* given in Fig. 6.3 is a variation of the algorithm in Fig. 4.5 that instead returns a single predicate. For this algorithm, it is assumed that the predicate rewrite system $\rightarrowtail$ is monotone (as required by Proposition 4.5.3 and Proposition 4.5.4) and the regularisation $\overline{\rightarrowtail}$ of $\rightarrowtail$ is a predicate rewrite system that is separable,

descending, and switchable (as required by Proposition 4.6.10). Even if these conditions are not satisfied, the algorithm is sound; that is, only (regularisations of) expected predicates are generated.

```
function Predicate(ℰ, ↣, P) returns a predicate;
input: ℰ, a set of examples;
       ↣, a predicate rewrite system;
       P, prune parameter;

openList := [top];
predicate := top;
accuracy := A_ℰ;
while openList ≠ [] do
    p := head(openList);
    openList := tail(openList);
    for each LR redex r via r↣b, for some b, in p do
        q := p[r/b];
        if q is regular then
            𝒫 := binary partition of ℰ induced by q;
            if A_𝒫 > accuracy then
                predicate := q;
                accuracy := A_𝒫;
                if A_𝒫 > P then P := A_𝒫;
            if B_𝒫 ≥ P ∧ A_𝒫 < B_𝒫 then
                openList := Insert(q, openList);
return predicate;
```

Fig. 6.3. Algorithm for finding a predicate to split a node

The meaning of the parameter *prune* is as follows.

prune: This parameter is a percentage. Only predicates that induce a partition with a refinement bound greater than or equal to *prune* are put on the open list. The default value is 0%.

The algorithm prunes predicates whose induced partitions have a refinement bound smaller than *prune* and thus removes predicates that do not have the potential for achieving splits of high accuracy. In the default mode, the parameter is initially set at 0%; as better accuracies are obtained during the search, the value of the parameter is updated. By Proposition 6.1.1, this kind of pruning is safe. However, for large search spaces, it is common to set a high initial value for *prune* and this may result in pruning that is not safe.

The open list is decreasingly ordered by the refinement bounds of the partitions induced by the predicates. (Predicates that have the same refine-

ment bound are decreasingly ordered by accuracy.) Thus the predicate with the highest such value is at the head of the list. The refinement bound plays an important part in directing the search towards promising predicates, that is, those which have the potential for being strengthened to produce splits with high accuracy. In this regard, in Fig. 6.3, the function *Insert* takes a predicate and the open list as arguments and returns a new open list with the predicate inserted in the appropriate place according to the ordering imposed by refinement bound.

ALKEMY has some other parameters that will be used in Sect. 6.2.

> *stump*: This parameter is a boolean. If *stump* is *true*, then the induced decision tree is a stump, that is, has a single split. If *stump* is *false*, then the induced decision tree can have any number of splits. The default value is *false*.
>
> *cutout*: This parameter is a non-negative integer. If the algorithm investigates successively *cutout* predicates without finding one which strictly improves the current best accuracy, then the algorithm terminates, returning the best predicate found so far. If *cutout* is 0, then no such limit is set. The default value is 0.

This completes the description of the decision-tree algorithm.

The final topic in this section before moving on to the illustrations in the next section is that of a methodology for using the tools introduced in this book. What follows is an outline of such a methodology.

The first step towards solving a learning problem is to determine a suitable type to model the individuals. As has been stressed already, establishing the type of the individuals is important because many of the transformations (the generic ones) depend only on this type. Having established this type, the individuals can be represented and the examples collected.

The second step is to write down the transformations that are appropriate for the learning problem. Useful collections of generic transformations determined by the type of the individuals are given in Chap. 4. A possibly much more difficult task is to choose a collection of domain-specific transformations. This may require expert knowledge of the problem domain.

The third step is an iterative one of finding a suitable hypothesis language. Using the approach of this book, this is equivalent to finding a suitable predicate rewrite system. Typically finding a suitable hypothesis language requires considerable experimentation and there is not much that one can say in general about it. However, here are two remarks. The first is that what ALKEMY is good at is the combinatorial task of combining the *basic ingredients* of a good hypothesis into the hypothesis itself. However, ALKEMY must have access to these basic ingredients in the form of appropriate transformations in the predicate rewrite system – if the basic ingredients are not present, it is unreasonable to expect the system to be able to invent them itself. Assuming the basic ingredients are present, ALKEMY is an excellent tool for combining

them in unexpected ways to produce an insightful hypothesis. The second remark is that what varies in each experiment is the predicate rewrite system – the representation of the individuals remains fixed (unless, of course, it is realised that there is a flaw in the representation and it should be revisited). The effects of the traditional methods of feature construction, extraction and selection are accomplished here solely by adjusting the predicate rewrite system in an appropriate way. A helpful way to understand this is to regard the conventional approach of working with feature vectors as a 'compiled' version of the approach here – instead of having the features (that is, predicates) given by the predicate rewrite system, they are pushed back into the representation of the individuals themselves. Working directly with feature vectors can be much more efficient, but conceptually it is clearer to work with predicate rewrite systems. If efficiency becomes a major issue, ALKEMY of course allows one to work with feature vectors in the traditional way.

The final step is that of evaluating the induced hypothesis. This may involve estimating the error rate or cost-sensitive loss function on a previously unseen test set. In the case of knowledge discovery, it may involve evaluating the impact of (some part of) the induced hypothesis in the application domain.

6.2 Illustrations

This section contains a series of illustrations that serve to elucidate the ideas of this book. The illustrations are deliberately simple ones and are varied, so that the approach to the representation of individuals and predicate construction can be more easily comprehended. For each illustration, the induced hypothesis given in the text was obtained by simply using the entire set of examples as training examples (and, for all but the Musk and Mutagenesis illustrations, the default settings for ALKEMY). No accuracies of the induced hypotheses are indicated, although the methods have been shown elsewhere to be competitive in this regard. Instead I concentrate on the representation of the structured individuals, the flexibility of predicate rewrite systems, and the comprehensibility of the induced hypotheses. It is also worth noting that for each illustration there are generally many predicate rewrite systems that could be chosen. I have made just one choice for each illustration; readers are encouraged to obtain the ALKEMY system and experiment with other possibilities.

Tennis

This illustration is a typical attribute-value problem which involves learning the concept of playing, or not playing, tennis, according to the weather. First the types *Outlook*, *Temperature*, *Humidity*, and *Wind* are introduced as follows.

$$\begin{aligned}
&Sunny, Overcast, Rain : Outlook\\
&Hot, Mild, Cool : Temperature\\
&High, Normal, Low : Humidity\\
&Strong, Medium, Weak : Wind.
\end{aligned}$$

It will be convenient to introduce the following type synonym.

$$Weather = Outlook \times Temperature \times Humidity \times Wind.$$

The function *playTennis* to be learned has signature

$$playTennis : Weather \rightarrow \Omega.$$

Here are the examples.

$$\begin{aligned}
&playTennis\ (Overcast, Hot, High, Weak) = \top\\
&playTennis\ (Rain, Mild, High, Weak) = \top\\
&playTennis\ (Rain, Cool, Normal, Weak) = \top\\
&playTennis\ (Overcast, Cool, Normal, Strong) = \top\\
&playTennis\ (Sunny, Cool, Normal, Weak) = \top\\
&playTennis\ (Rain, Mild, Normal, Weak) = \top\\
&playTennis\ (Sunny, Mild, Normal, Strong) = \top\\
&playTennis\ (Overcast, Mild, High, Strong) = \top\\
&playTennis\ (Overcast, Hot, Normal, Weak) = \top\\
&playTennis\ (Sunny, Hot, High, Weak) = \bot\\
&playTennis\ (Sunny, Hot, High, Strong) = \bot\\
&playTennis\ (Rain, Cool, Normal, Strong) = \bot\\
&playTennis\ (Sunny, Mild, High, Weak) = \bot\\
&playTennis\ (Rain, Mild, High, Strong) = \bot.
\end{aligned}$$

The background theory contains the following transformations. (In this and later illustrations, the inclusion of the predicate $top : a \rightarrow \Omega$ in the background theory is taken for granted.)

$$\begin{aligned}
&(= Sunny) : Outlook \rightarrow \Omega\\
&(= Overcast) : Outlook \rightarrow \Omega\\
&(= Rain) : Outlook \rightarrow \Omega\\
&(= Hot) : Temperature \rightarrow \Omega\\
&(= Mild) : Temperature \rightarrow \Omega\\
&(= Cool) : Temperature \rightarrow \Omega\\
&(= High) : Humidity \rightarrow \Omega
\end{aligned}$$

$$(= Normal) : Humidity \to \Omega$$
$$(= Low) : Humidity \to \Omega$$
$$(= Strong) : Wind \to \Omega$$
$$(= Medium) : Wind \to \Omega$$
$$(= Weak) : Wind \to \Omega$$
$$projOutlook : Weather \to Outlook$$
$$projTemperature : Weather \to Temperature$$
$$projHumidity : Weather \to Humidity$$
$$projWind : Weather \to Wind$$
$$\wedge_2 : (Weather \to \Omega) \to (Weather \to \Omega) \to Weather \to \Omega.$$

Here *projOutlook* is the projection from *Weather* onto the component *Outlook*. Similarly, for *projTemperature*, *projHumidity*, and *projWind*.

The last transformation above is noteworthy. To explain why, consider the transformation and_2 defined as follows.

$$and_2 : (Weather \to \Omega) \to (Weather \to \Omega) \to Weather \to \Omega$$

$$and_2 = \wedge_2.$$

In this definition, the transformation $\wedge_2 : (a \to \Omega) \to (a \to \Omega) \to a \to \Omega$ is the one that was defined in Chap. 4. Thus and_2 is a new transformation that is, in effect, a specialised version of $\wedge_2$ that only applies to predicates of type $Weather \to \Omega$. Rather than explicitly introduce specialised transformations like this, since the intention should be clear I simply use the name of the general transformation in the background theory, but give it an appropriately specialised signature.

A suitable predicate rewrite system for this illustration is as follows.

$$top \rightarrowtail \wedge_2\ top\ top$$
$$top \rightarrowtail projOutlook \circ top$$
$$top \rightarrowtail projTemperature \circ top$$
$$top \rightarrowtail projHumidity \circ top$$
$$top \rightarrowtail projWind \circ top$$
$$top \rightarrowtail (= Sunny)$$
$$top \rightarrowtail (= Overcast)$$
$$top \rightarrowtail (= Rain)$$
$$top \rightarrowtail (= Hot)$$
$$top \rightarrowtail (= Mild)$$
$$top \rightarrowtail (= Cool)$$
$$top \rightarrowtail (= High)$$

$$
\begin{aligned}
&top \rightarrowtail (= Normal)\\
&top \rightarrowtail (= Low)\\
&top \rightarrowtail (= Strong)\\
&top \rightarrowtail (= Medium)\\
&top \rightarrowtail (= Weak).
\end{aligned}
$$

For this setting, ALKEMY found the following definition for the function *playTennis*.

$$
\begin{aligned}
&playTennis\ w =\\
&\quad if\ \wedge_2\ (projOutlook \circ (= Sunny))\ (projHumidity \circ (= High))\ w\\
&\quad then\ \bot\\
&\quad else\ if\ \wedge_2\ (projOutlook \circ (= Rain))\ (projWind \circ (= Strong))\ w\\
&\qquad then\ \bot\\
&\qquad else\ \top.
\end{aligned}
$$

"The weather is suitable for playing tennis unless the outlook is sunny and the humidity is high or the outlook is rain and the wind is strong".

Other predicate rewrite systems can also be used to give an equivalent hypothesis. For example, one can look for three or four conditions on an individual. For four conditions, the top-level rewrite becomes

$$top \rightarrowtail \wedge_4\ top\ top\ top\ top.$$

Alternatively, one can use the more precise top-level rewrite

$$
\begin{aligned}
top \rightarrowtail \wedge_4\ (projOutlook \circ top)\ &(projTemperature \circ top)\\
&(projHumidity \circ top)\ (projWind \circ top).
\end{aligned}
$$

Keys

I return to the problem that was employed in Sect. 1.3 to motivate the learning aspects of this book. This problem is interesting because it illustrates what is known as a 'multiple-instance' problem. From a knowledge representation point of view, a multiple-instance problem is indicated by having a term which is a set at the top level representing an individual.

Here is the statement of the problem again. Consider the problem of determining whether a key in a bunch of keys can open a door. More precisely, suppose there are a number of bunches of keys and a particular door which can be opened by a key. For each bunch of keys either no key opens the door or there is at least one key which opens the door. For each bunch of keys it is known whether there is some key which opens the door, but it is not known precisely which key does the job, or it is known that no key opens the door.

The problem is to find a classification function for the bunches of keys, where the classification is into those which contain a key that opens the door and those that do not.

This problem is prototypical of a number of important practical problems such as that of pharmacophore discovery. A characteristic property of the multiple-instance problem is that the individuals are actually a collection of similar entities and the problem is to try to find a suitable entity in each collection satisfying some predicate. Thus the kind of predicate one wants to find has the form

$$(setExists_1\ p)$$

for a suitable predicate p on the entities in the collection. Expressed this way, it is clear that the multiple-instance problem is just a special case of the general framework that is developed in this book in which the type of the individuals is a set. Since the individuals to be classified have a set type, following the principles proposed here, one should go inside the set to apply predicates to its elements.

Here are some types and constants to model the individuals.

$$Abloy, Chubb, Rubo, Yale : Make$$
$$Short, Medium, Long : Length$$
$$Narrow, Normal, Broad : Width.$$

The following type synonyms are introduced.

$$NumProngs = Nat$$
$$Key = Make \times NumProngs \times Length \times Width$$
$$Bunch = \{Key\}.$$

The function *opens* to be learned has signature

$$opens : Bunch \rightarrow \Omega.$$

The examples are as follows.

$$opens\ \{(Abloy, 3, Short, Normal), (Abloy, 4, Medium, Broad), (Chubb, 3, Long, Narrow)\} = \top$$
$$opens\ \{(Abloy, 3, Medium, Broad), (Chubb, 2, Long, Normal), (Chubb, 4, Medium, Broad)\} = \top$$
$$opens\ \{(Abloy, 3, Short, Broad), (Abloy, 4, Medium, Broad), (Chubb, 3, Long, Narrow)\} = \top$$
$$opens\ \{(Abloy, 3, Medium, Broad), (Abloy, 4, Medium, Narrow), (Chubb, 3, Long, Broad), (Yale, 4, Medium, Broad)\} = \top$$

$$
\begin{aligned}
&opens\ \{(Abloy, 3, Medium, Narrow), (Chubb, 6, Medium, Normal),\\
&\qquad (Rubo, 5, Short, Narrow), (Yale, 4, Long, Broad)\} = \top\\
&opens\ \{(Chubb, 3, Short, Broad), (Chubb, 4, Medium, Broad),\\
&\qquad (Yale, 3, Short, Narrow), (Yale, 4, Long, Normal)\} = \bot\\
&opens\ \{(Yale, 3, Long, Narrow), (Yale, 4, Long, Broad)\} = \bot\\
&opens\ \{(Abloy, 3, Short, Broad), (Chubb, 3, Short, Broad),\\
&\qquad (Rubo, 4, Long, Broad), (Yale, 4, Long, Broad)\} = \bot\\
&opens\ \{(Abloy, 4, Short, Broad), (Chubb, 3, Medium, Broad),\\
&\qquad (Rubo, 5, Long, Narrow)\} = \bot.
\end{aligned}
$$

The background theory contains the following transformations.

$$
\begin{aligned}
&(= Abloy) : Make \to \Omega\\
&(= Chubb) : Make \to \Omega\\
&(= Rubo) : Make \to \Omega\\
&(= Yale) : Make \to \Omega\\
&(= 2) : NumProngs \to \Omega\\
&(= 3) : NumProngs \to \Omega\\
&(= 4) : NumProngs \to \Omega\\
&(= 5) : NumProngs \to \Omega\\
&(= 6) : NumProngs \to \Omega\\
&(= Short) : Length \to \Omega\\
&(= Medium) : Length \to \Omega\\
&(= Long) : Length \to \Omega\\
&(= Narrow) : Width \to \Omega\\
&(= Normal) : Width \to \Omega\\
&(= Broad) : Width \to \Omega\\
&projMake : Key \to Make\\
&projNumProngs : Key \to NumProngs\\
&projLength : Key \to Length\\
&projWidth : Key \to Width\\
&setExists_1 : (Key \to \Omega) \to Bunch \to \Omega\\
&\wedge_2 : (Key \to \Omega) \to (Key \to \Omega) \to Key \to \Omega\\
&\wedge_3 : (Key \to \Omega) \to (Key \to \Omega) \to (Key \to \Omega) \to Key \to \Omega\\
&\wedge_4 : (Key \to \Omega) \to (Key \to \Omega) \to (Key \to \Omega) \to (Key \to \Omega) \to Key \to \Omega.
\end{aligned}
$$

Here is the predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail setExists_1\ (\wedge_4\ (projMake \circ top)\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ top)\ (projWidth \circ top))\\
&top \rightarrowtail (= Abloy)\\
&top \rightarrowtail (= Chubb)\\
&top \rightarrowtail (= Rubo)\\
&top \rightarrowtail (= Yale)\\
&top \rightarrowtail (= 2)\\
&top \rightarrowtail (= 3)\\
&top \rightarrowtail (= 4)\\
&top \rightarrowtail (= 5)\\
&top \rightarrowtail (= 6)\\
&top \rightarrowtail (= Short)\\
&top \rightarrowtail (= Medium)\\
&top \rightarrowtail (= Long)\\
&top \rightarrowtail (= Narrow)\\
&top \rightarrowtail (= Normal)\\
&top \rightarrowtail (= Broad).
\end{aligned}
$$

For this setting, ALKEMY found the following definition for the function *opens*.

$$
\begin{aligned}
&opens\ b =\\
&\quad if\ setExists_1\ (\wedge_4\ (projMake \circ (= Abloy))\ (projNumProngs \circ top)\\
&\qquad\qquad (projLength \circ (= Medium))\ (projWidth \circ top))\ b\\
&\quad then\ \top\\
&\quad else\ \bot.
\end{aligned}
$$

"A bunch of keys opens the door if and only if it contains an Abloy key of medium length".

The definition for *opens* can be simplified to

$$
\begin{aligned}
opens\ b = setExists_1\ (\wedge_2\ &(projMake \circ (= Abloy))\\
&(projLength \circ (= Medium)))\ b.
\end{aligned}
$$

Climate

I now consider an illustration concerning the use of multisets for knowledge representation.

Consider the problem of trying to decide whether a climate in some country is pleasant or not. The climate is modelled by a multiset. Each item in

a multiset is a 4-tuple characterising the main features of the weather during a day and the multiplicity of the item is the number of times during a year a day with those particular weather features occurs. Thus, using the type *Weather* from the tennis illustration, a climate can be modelled by the following multiset type.

$$Climate = Weather \rightarrow Nat.$$

The function *pleasant* to be learned has signature

$$pleasant : Climate \rightarrow \Omega.$$

Suppose the examples are as follows.

$$\begin{aligned}
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, Normal, Strong)\ then\ 220\ else\\
&\qquad\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 15\ else\\
&\qquad\qquad if\ w = (Sunny, Hot, High, Strong)\ then\ 100\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 30\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Strong)\ then\ 180\ else\\
&\qquad\qquad if\ w = (Sunny, Mild, High, Weak)\ then\ 25\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ 160\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Strong)\ then\ 135\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 130\ else\\
&\qquad\qquad if\ w = (Sunny, Mild, Normal, Weak)\ then\ 100\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, Normal, Weak)\ then\ 150\ else\\
&\qquad\qquad if\ w = (Sunny, Mild, High, Weak)\ then\ 215\ else\ 0) = \top\\
&pleasant\ (\lambda w.if\ w = (Overcast, Hot, High, Weak)\ then\ 99\ else\\
&\qquad\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 266\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Overcast, Hot, High, Weak)\ then\ 35\ else\\
&\qquad\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 124\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 206\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, High, Weak)\ then\ 160\ else\\
&\qquad\qquad if\ w = (Overcast, Cool, Normal, Strong)\ then\ 50\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ 155\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Sunny, Mild, High, Weak)\ then\ 135\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Strong)\ then\ 34\ else\\
&\qquad\qquad if\ w = (Sunny, Hot, High, Strong)\ then\ 196\ else\ 0) = \bot\\
&pleasant\ (\lambda w.if\ w = (Rain, Mild, High, Strong)\ then\ 100\ else\\
&\qquad\qquad if\ w = (Overcast, Hot, Normal, Weak)\ then\ 80\ else\\
&\qquad\qquad if\ w = (Rain, Cool, Normal, Weak)\ then\ 185\ else\ 0) = \bot.
\end{aligned}$$

The background theory contains the following transformations.

$$
\begin{aligned}
&(= Sunny) : Outlook \to \Omega \\
&\qquad\qquad \vdots \\
&(= Weak) : Wind \to \Omega \\
&projOutlook : Weather \to Outlook \\
&projTemperature : Weather \to Temperature \\
&projHumidity : Weather \to Humidity \\
&projWind : Weather \to Wind \\
&(> 0) : Nat \to \Omega \\
&(> 50) : Nat \to \Omega \\
&(> 100) : Nat \to \Omega \\
&\qquad\qquad \vdots \\
&(> 350) : Nat \to \Omega \\
&domMcard : (Weather \to \Omega) \to Climate \to Nat \\
&\wedge_4 : (Weather \to \Omega) \to (Weather \to \Omega) \to (Weather \to \Omega) \\
&\qquad\qquad \to (Weather \to \Omega) \to Weather \to \Omega.
\end{aligned}
$$

Here is the predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail (domMcard\ top) \circ (> 0) \\
&top \rightarrowtail \wedge_4\ (projOutlook \circ top)\ (projTemperature \circ top) \\
&\qquad\qquad (projHumidity \circ top)\ (projWind \circ top) \\
&top \rightarrowtail (= Sunny) \\
&top \rightarrowtail (= Overcast) \\
&top \rightarrowtail (= Rain) \\
&top \rightarrowtail (= Hot) \\
&top \rightarrowtail (= Mild) \\
&top \rightarrowtail (= Cool) \\
&top \rightarrowtail (= High) \\
&top \rightarrowtail (= Normal) \\
&top \rightarrowtail (= Low) \\
&top \rightarrowtail (= Strong) \\
&top \rightarrowtail (= Medium) \\
&top \rightarrowtail (= Weak) \\
&(> 0) \rightarrowtail (> 50) \\
&(> 50) \rightarrowtail (> 100)
\end{aligned}
$$

$$(> 100) \rightarrowtail (> 150)$$
$$(> 150) \rightarrowtail (> 200)$$
$$(> 200) \rightarrowtail (> 250)$$
$$(> 250) \rightarrowtail (> 300)$$
$$(> 300) \rightarrowtail (> 350).$$

For this setting, ALKEMY found the following definition for the function *pleasant*.

$$\begin{aligned}
&\mathit{pleasant}\ c = \\
&\quad \mathit{if}\ (\mathit{domMcard}\ (\wedge_4\ (\mathit{projOutlook} \circ (= \mathit{Sunny}))\ (\mathit{projTemperature} \circ (= \mathit{Mild})) \\
&\qquad\qquad (\mathit{projHumidity} \circ \mathit{top})\ (\mathit{projWind} \circ\ = \mathit{top}))) \circ (> 150)\ c \\
&\quad \mathit{then}\ \top \\
&\quad \mathit{else}\ \bot.
\end{aligned}$$

"A country has a pleasant climate if and only if it has at least 150 days in a year that have a sunny outlook and a mild temperature".

Trains

The next illustration is the well-known East–West Challenge, which consists of determining whether a train is headed west or east based on a number of its characteristics. The trains are illustrated in Fig. 6.4.

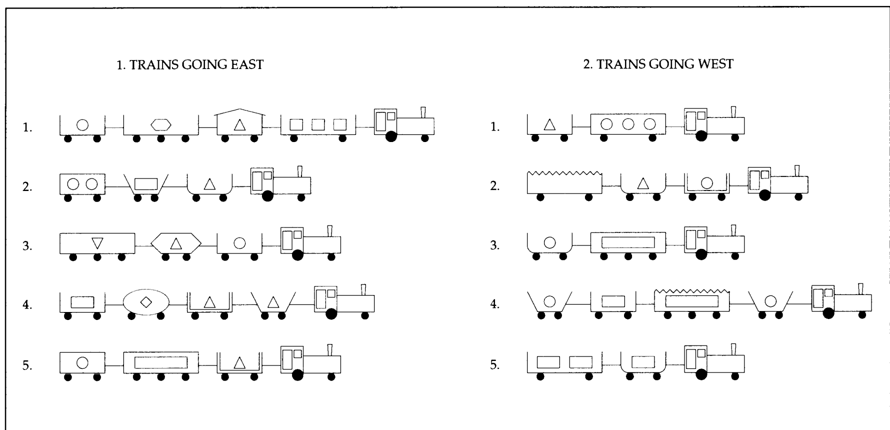

Fig. 6.4. Trains going east and west

The most natural type to model a train is a list. I first introduce the types *Direction*, *Shape*, *Length*, *Kind*, *Roof*, and *Object*.

$$East, West : Direction$$
$$Rectangular, DoubleRectangular, UShaped, BucketShaped, \\ Hexagonal, Ellipsoidal : Shape$$
$$Long, Short : Length$$
$$Closed, Open : Kind$$
$$Flat, Jagged, Peaked, Curved, None : Roof$$
$$Circle, Hexagon, Square, Rectangle, LongRectangle, Triangle, \\ InvertedTriangle, Diamond, Null : Object.$$

It will be convenient to introduce the following type synonyms.

$$NumWheels = Nat$$
$$NumObjects = Nat$$
$$Load = Object \times NumObjects$$
$$Car = Shape \times Length \times NumWheels \times Kind \times Roof \times Load$$
$$Train = List\ Car.$$

The function *direction* to be learned has signature

$$direction : Train \rightarrow Direction.$$

Here are the examples. (See Fig. 6.4.)

$$\begin{aligned}
direction\ &[(Rectangular, Long, 2, Open, None, (Square, 3)), \\
&(Rectangular, Short, 2, Closed, Peaked, (Triangle, 1)), \\
&(Rectangular, Long, 3, Open, None, (Hexagon, 1)), \\
&(Rectangular, Short, 2, Open, None, (Circle, 1))] = East \\
direction\ &[(UShaped, Short, 2, Open, None, (Triangle, 1)), \\
&(BucketShaped, Short, 2, Open, None, (Rectangle, 1)), \\
&(Rectangular, Short, 2, Closed, Flat, (Circle, 2))] = East \\
direction\ &[(Rectangular, Short, 2, Open, None, (Circle, 1)), \\
&(Hexagonal, Short, 2, Closed, Flat, (Triangle, 1)), \\
&(Rectangular, Long, 3, Closed, Flat, (InvertedTriangle, 1))] = East \\
direction\ &[(BucketShaped, Short, 2, Open, None, (Triangle, 1)), \\
&(DoubleRectangular, Short, 2, Open, None, (Triangle, 1)), \\
&(Ellipsoidal, Short, 2, Closed, Curved, (Diamond, 1)), \\
&(Rectangular, Short, 2, Open, None, (Rectangle, 1))] = East \\
direction\ &[(DoubleRectangular, Short, 2, Open, None, (Triangle, 1)), \\
&(Rectangular, Long, 3, Closed, Flat, (LongRectangle, 1)), \\
&(Rectangular, Short, 2, Closed, Flat, (Circle, 1))] = East
\end{aligned}$$

$$
\begin{aligned}
&\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Closed}, \mathit{Flat}, (\mathit{Circle}, 3)),\\
&\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1))] = \mathit{West}\\
&\mathit{direction}\ [(\mathit{DoubleRectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),\\
&\qquad (\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Triangle}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Closed}, \mathit{Jagged}, (\mathit{Null}, 0))] = \mathit{West}\\
&\mathit{direction}\ [(\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Flat}, (\mathit{LongRectangle}, 1)),\\
&\qquad (\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{West}\\
&\mathit{direction}\ [(\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Long}, 3, \mathit{Closed}, \mathit{Jagged}, (\mathit{LongRectangle}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
&\qquad (\mathit{BucketShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Circle}, 1))] = \mathit{West}\\
&\mathit{direction}\ [(\mathit{UShaped}, \mathit{Short}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 1)),\\
&\qquad (\mathit{Rectangular}, \mathit{Long}, 2, \mathit{Open}, \mathit{None}, (\mathit{Rectangle}, 2))] = \mathit{West}.
\end{aligned}
$$

The background theory contains the following transformations.

$$
\begin{aligned}
&(= \mathit{Rectangular}) : \mathit{Shape} \to \Omega\\
&\qquad \vdots\\
&(= \mathit{Null}) : \mathit{Object} \to \Omega\\
&\mathit{projShape} : \mathit{Car} \to \mathit{Shape}\\
&\mathit{projLength} : \mathit{Car} \to \mathit{Length}\\
&\mathit{projNumWheels} : \mathit{Car} \to \mathit{NumWheels}\\
&\mathit{projKind} : \mathit{Car} \to \mathit{Kind}\\
&\mathit{projRoof} : \mathit{Car} \to \mathit{Roof}\\
&\mathit{projLoad} : \mathit{Car} \to \mathit{Load}\\
&\mathit{projObject} : \mathit{Load} \to \mathit{Object}\\
&\mathit{projNumObjects} : \mathit{Load} \to \mathit{NumObjects}\\
&\mathit{null} : \mathit{Train} \to \Omega\\
&\mathit{head} : \mathit{Train} \to \mathit{Car}\\
&\mathit{last} : \mathit{Train} \to \mathit{Car}\\
&\mathit{listToSet} : \mathit{Train} \to \{\mathit{Car}\}\\
&(\mathit{sublists}\ 2) : \mathit{Train} \to \{\mathit{Train}\}\\
&\mathit{setExists}_1 : (\mathit{Car} \to \Omega) \to \{\mathit{Car}\} \to \Omega\\
&\mathit{setExists}_1 : (\mathit{Train} \to \Omega) \to \{\mathit{Train}\} \to \Omega\\
&(!!0) : \mathit{Train} \to \mathit{Car}\\
&(!!1) : \mathit{Train} \to \mathit{Car}\\
&\wedge_2 : (\mathit{Car} \to \Omega) \to (\mathit{Car} \to \Omega) \to \mathit{Car} \to \Omega
\end{aligned}
$$

$$\wedge_2 : (\mathit{Train} \to \Omega) \to (\mathit{Train} \to \Omega) \to \mathit{Train} \to \Omega$$

Here is the predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail listToSet \circ (setExists_1\ (\wedge_2\ top\ top))\\
&top \rightarrowtail (sublists\ 2) \circ (setExists_1\ (\wedge_2\ ((!!0) \circ top)\ ((!!1) \circ top)))\\
&top \rightarrowtail projShape \circ top\\
&top \rightarrowtail projLength \circ top\\
&top \rightarrowtail projNumWheels \circ top\\
&top \rightarrowtail projKind \circ top\\
&top \rightarrowtail projRoof \circ top\\
&top \rightarrowtail projLoad \circ top\\
&top \rightarrowtail projObject \circ top\\
&top \rightarrowtail projNumObjects \circ top\\
&top \rightarrowtail (=Circle)\\
&top \rightarrowtail (=Hexagon)\\
&top \rightarrowtail (=Square)\\
&top \rightarrowtail (=Rectangle)\\
&top \rightarrowtail (=LongRectangle)\\
&top \rightarrowtail (=Triangle)\\
&top \rightarrowtail (=UTriangle)\\
&top \rightarrowtail (=Diamond)\\
&top \rightarrowtail (=Null)\\
&top \rightarrowtail (=Flat)\\
&top \rightarrowtail (=Jagged)\\
&top \rightarrowtail (=Peaked)\\
&top \rightarrowtail (=Curved)\\
&top \rightarrowtail (=None)\\
&top \rightarrowtail (=Rectangular)\\
&top \rightarrowtail (=DoubleRectangular)\\
&top \rightarrowtail (=UShaped)\\
&top \rightarrowtail (=BucketShaped)\\
&top \rightarrowtail (=Hexagonal)\\
&top \rightarrowtail (=Ellipsoidal)\\
&top \rightarrowtail (=Long)\\
&top \rightarrowtail (=Short)\\
&top \rightarrowtail (=Closed)
\end{aligned}
$$

$$top \rightarrowtail (= Open)$$
$$top \rightarrowtail (= 2)$$
$$top \rightarrowtail (= 3).$$

The first rewrite above generates a condition on the set of cars in the train. The second rewrite generates conditions on two contiguous cars.

For this setting, ALKEMY found the following definition for the function *direction*.

$$\begin{aligned}
&direction\ t = \\
&\quad if\ listToSet \circ (setExists_1\ (\wedge_2\ (projLength \circ (= Short)) \\
&\qquad\qquad\qquad\qquad (projKind \circ (= Closed))))\ t \\
&\quad then\ East \\
&\quad else\ West.
\end{aligned}$$

"A train is eastbound if and only if it has a short closed car".

Bongard

The next illustration is problem 47 in the book by Bongard on pattern recognition. This problem is illustrated in Fig. 6.5.

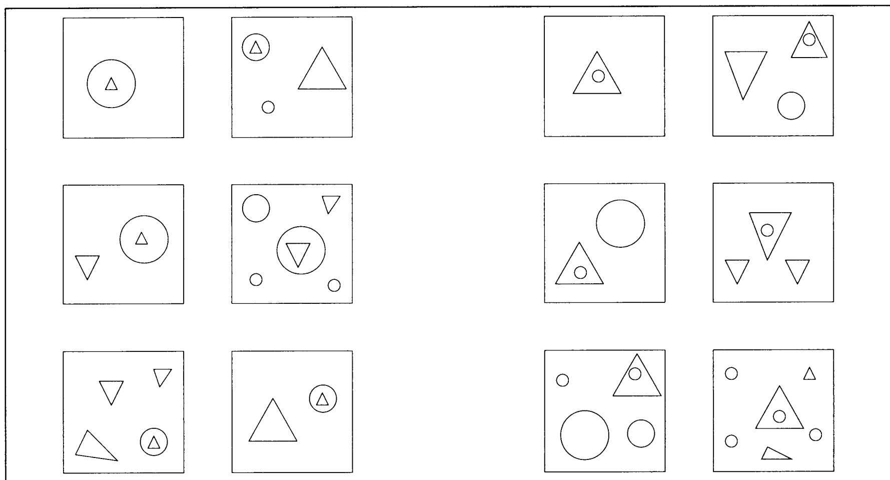

Fig. 6.5. Bongard problem 47: Examples in *Class1* are on the left; those in *Class2* on the right

As usual, one must first decide on a suitable type to represent a Bongard diagram. For this, I have chosen to use a digraph. The geometric figures in

the diagram, which can be circles, triangles, or squares, are the nodes of the graph and there is a directed edge from shape s_1 to shape s_2 iff s_2 is located inside s_1.

The types *Shape* and *Class* are introduced as follows.

$$Circle, Triangle, Square : Shape$$
$$Class1, Class2 : Class.$$

In this illustration, there is no information on the edges of the graph, so the type *1* is used as a default for the type of this information and the empty tuple () is used for all these values. The type *Diagram* is defined as follows.

$$Diagram = Digraph\ Shape\ 1.$$

Vertices contain information of type *Shape*, while edges contain information of type *1* (which is equivalent to having no information at all). Thus the term

$$(\{(1, Circle), (2, Triangle)\}, \{((1,2), ())\})$$

represents the diagram with a circle and a triangle, and where the triangle is inside the circle.

The function *bongard* to be learned has signature

$$bongard : Diagram \rightarrow Class.$$

Here are the examples.

$$bongard\ (\{(1, Circle), (2, Triangle)\}, \{((1,2), ())\}) = Class1$$
$$bongard\ (\{(1, Circle), (2, Triangle), (3, Triangle), (4, Circle)\},$$
$$\{((1,2), ())\}) = Class1$$
$$bongard\ (\{(1, Triangle), (2, Circle), (3, Triangle)\}, \{((2,3), ())\}) = Class1$$
$$bongard\ (\{(1, Circle), (2, Triangle), (3, Circle), (4, Triangle)$$
$$(5, Circle), (6, Circle)\}, \{((3,4), ())\}) = Class1$$
$$bongard\ (\{(1, Triangle), (2, Triangle), (3, Triangle), (4, Circle),$$
$$(5, Triangle)\}, \{((4,5), ())\}) = Class1$$
$$bongard\ (\{(1, Triangle), (2, Circle), (3, Triangle)\}, \{((2,3), ())\}) = Class1$$
$$bongard\ (\{(1, Triangle), (2, Circle)\}, \{((1,2), ())\}) = Class2$$
$$bongard\ (\{(1, Triangle), (2, Triangle), (3, Circle), (4, Circle)\},$$
$$\{((2,3), ())\}) = Class2$$
$$bongard\ (\{(1, Circle), (2, Triangle), (3, Circle)\}, \{((2,3), ())\}) = Class2$$
$$bongard\ (\{(1, Triangle), (2, Circle), (3, Triangle), (4, Triangle)\},$$
$$\{((1,2), ())\}) = Class2$$

$$bongard\ (\{(1, Circle), (2, Triangle), (3, Circle), (4, Circle), (5, Circle)\},$$
$$\{((2,3),())\}) = Class2$$
$$bongard\ (\{(1, Circle), (2, Triangle), (3, Triangle), (4, Circle),$$
$$(5, Circle), (6, Triangle), (7, Circle)\}, \{((3,4),())\}) = Class2.$$

The background theory contains the following transformations.

$$(= Circle) : Shape \to \Omega$$
$$(= Triangle) : Shape \to \Omega$$
$$(= Square) : Shape \to \Omega$$
$$vertices : Diagram \to \{DiVertex\ Shape\ 1\}$$
$$edges : Diagram \to \{DiEdge\ Shape\ 1\}$$
$$vertex : DiVertex\ Shape\ 1 \to Shape$$
$$edge : DiEdge\ Shape\ 1 \to 1$$
$$connects : DiEdge\ Shape\ 1 \to DiVertex\ Shape\ 1 \times DiVertex\ Shape\ 1$$
$$projDiVertex1 : DiVertex\ Shape\ 1 \times DiVertex\ Shape\ 1$$
$$\to DiVertex\ Shape\ 1$$
$$projDiVertex2 : DiVertex\ Shape\ 1 \times DiVertex\ Shape\ 1$$
$$\to DiVertex\ Shape\ 1$$
$$setExists_1 : (DiVertex\ Shape\ 1 \to \Omega) \to \{DiVertex\ Shape\ 1\} \to \Omega$$
$$setExists_1 : (DiEdge\ Shape\ 1 \to \Omega) \to \{DiEdge\ Shape\ 1\} \to \Omega$$
$$\wedge_2 : (DiVertex\ Shape\ 1 \to \Omega) \to (DiVertex\ Shape\ 1 \to \Omega)$$
$$\to DiVertex\ Shape\ 1 \to \Omega.$$

Here *projDiVertex1* projects onto the first component and *projDiVertex2* projects onto the second.

Here is the predicate rewrite system.

$$top \rightarrowtail edges \circ (setExists_1\ top)$$
$$top \rightarrowtail connects \circ top$$
$$top \rightarrowtail projDiVertex1 \circ vertex \circ top$$
$$top \rightarrowtail projDiVertex2 \circ vertex \circ top$$
$$top \rightarrowtail vertices \circ (setExists_1\ (vertex \circ top))$$
$$top \rightarrowtail (= Circle)$$
$$top \rightarrowtail (= Triangle)$$
$$top \rightarrowtail (= Square).$$

For this setting, Alkemy found the following definition for the function *bongard*.

$$\begin{aligned}&bongard\ d = \\&\quad if\ edges \circ (setExists_1\ (connects \circ projDiVertex1 \circ vertex \circ (= Circle)))\ d\\&\quad then\ Class1\\&\quad else\ Class2.\end{aligned}$$

"A diagram is in class 1 if it contains something inside a circle; otherwise, it is in class 2".

Chemicals

The next illustration involves learning a theory to predict whether a chemical molecule has a certain property.

As explained in Example 3.5.3, an undirected graph is used to model a molecule – an atom is a vertex in the graph and a bond is an edge. The type *Element* is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

I also make the following type synonyms.

$$\begin{aligned}&AtomType = Nat\\&Charge = Float\\&Atom = Element \times AtomType \times Charge\\&Bond = Nat.\end{aligned}$$

I now obtain the type of a molecule which is an (undirected) graph whose vertices have type *Atom* and whose edges have type *Bond*, which leads to the following definition.

$$Molecule = Graph\ Atom\ Bond.$$

The function *active* to be learned has signature

$$active : Molecule \to \Omega.$$

The examples are as follows.

$$\begin{aligned}
&\mathit{active}\ (\{(1,(C,22,-0.117)),(2,(O,45,-0.388)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\quad(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013))\},\\
&\quad\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),(\langle 3,4\rangle,7),\\
&\quad(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\quad(\langle 6,8\rangle,2),(\langle 11,12\rangle,7)\}) = \top\\
&\mathit{active}\ (\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\quad(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\quad(13,(C,22,-0.117))\},\\
&\quad\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\quad(\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \top\\
&\mathit{active}\ (\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\quad\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&\quad(\langle 6,8\rangle,2)\}) = \top\\
&\mathit{active}\ (\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(H,3,0.143)),(6,(N,38,0.812)),\\
&\quad(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&\quad(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&\quad(13,(C,22,-0.117))\},\\
&\quad\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&\quad(\langle 6,7\rangle,2),(\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \top\\
&\mathit{active}\ (\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(O,45,-0.338)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&\quad(7,(O,40,-0.388)),(8,(O,40,-0.388))\},\\
&\quad\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&\quad(\langle 3,4\rangle,7),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),(\langle 6,8\rangle,2)\}) = \top\\
&\mathit{active}\ (\{(1,(C,22,-0.117)),(2,(O,45,-0.388)),(3,(C,22,-0.117)),\\
&\quad(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),
\end{aligned}$$

$$
\begin{aligned}
&(7,(O,45,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),\\
&(\langle 6,7\rangle,2),(\langle 6,8\rangle,2),(\langle 11,12\rangle,7)\}) = \bot
\end{aligned}
$$

$$
\begin{aligned}
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(O,45,-0.338)),\\
&(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&(7,(O,40,-0.388)),(8,(O,40,-0.388))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&(\langle 3,4\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2)\}) = \bot\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&(7,(O,45,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,8\rangle,1),(\langle 3,9\rangle,1),\\
&(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),(\langle 6,8\rangle,3)\}) = \bot\\
\mathit{active}\ &(\{(1,(C,22,-0.117)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,35,0.812)),\\
&(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142)),\\
&(10,(H,3,0.142)),(11,(C,27,-0.087)),(12,(C,27,0.013)),\\
&(13,(C,22,-0.117))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 3,4\rangle,7),\\
&(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 4,11\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,2),\\
&(\langle 6,8\rangle,2),(\langle 11,12\rangle,7),(\langle 12,13\rangle,7)\}) = \bot\\
\mathit{active}\ &(\{(1,(H,3,0.142)),(2,(C,22,-0.117)),(3,(C,22,-0.117)),\\
&(4,(C,195,-0.087)),(5,(C,195,0.013)),(6,(N,38,0.812)),\\
&(7,(O,40,-0.388)),(8,(O,40,-0.388)),(9,(H,3,0.142))\},\\
&\{(\langle 1,2\rangle,7),(\langle 1,6\rangle,7),(\langle 1,7\rangle,1),(\langle 2,3\rangle,7),(\langle 2,8\rangle,1),\\
&(\langle 3,4\rangle,7),(\langle 3,9\rangle,1),(\langle 4,5\rangle,7),(\langle 5,6\rangle,7),(\langle 6,7\rangle,3),\\
&(\langle 6,8\rangle,2)\}) = \bot.
\end{aligned}
$$

The background theory contains the following transformations.

$$
\begin{aligned}
&(=Br) : \mathit{Element} \to \Omega\\
&\qquad\vdots\\
&(=S) : \mathit{Element} \to \Omega\\
&(=3) : \mathit{AtomType} \to \Omega
\end{aligned}
$$

$\vdots$

$(= 195) : \mathit{AtomType} \rightarrow \Omega$
$(\leq -0.117) : \mathit{Charge} \rightarrow \Omega$
$(\leq -0.087) : \mathit{Charge} \rightarrow \Omega$
$(\leq 0.013) : \mathit{Charge} \rightarrow \Omega$
$(\leq 0.142) : \mathit{Charge} \rightarrow \Omega$
$(\geq -0.117) : \mathit{Charge} \rightarrow \Omega$
$(\geq -0.087) : \mathit{Charge} \rightarrow \Omega$
$(\geq 0.013) : \mathit{Charge} \rightarrow \Omega$
$(\geq 0.142) : \mathit{Charge} \rightarrow \Omega$
$(= 1) : \mathit{Bond} \rightarrow \Omega$
$(= 2) : \mathit{Bond} \rightarrow \Omega$
$(= 3) : \mathit{Bond} \rightarrow \Omega$
$(= 7) : \mathit{Bond} \rightarrow \Omega$
$(> 0) : \mathit{Nat} \rightarrow \Omega$
$(> 1) : \mathit{Nat} \rightarrow \Omega$
$(> 2) : \mathit{Nat} \rightarrow \Omega$
$\mathit{projElement} : \mathit{Atom} \rightarrow \mathit{Element}$
$\mathit{projAtomType} : \mathit{Atom} \rightarrow \mathit{AtomType}$
$\mathit{projCharge} : \mathit{Atom} \rightarrow \mathit{Charge}$
$\mathit{vertices} : \mathit{Molecule} \rightarrow \{\mathit{Vertex\ Atom\ Bond}\}$
$\mathit{edges} : \mathit{Molecule} \rightarrow \{\mathit{Edge\ Atom\ Bond}\}$
$\mathit{vertex} : \mathit{Vertex\ Atom\ Bond} \rightarrow \mathit{Atom}$
$\mathit{connects} : \mathit{Edge\ Atom\ Bond} \rightarrow (\mathit{Vertex\ Atom\ Bond} \rightarrow \mathit{Nat})$
$\mathit{edge} : \mathit{Edge\ Atom\ Bond} \rightarrow \mathit{Bond}$
$\mathit{domCard} : (\mathit{Vertex\ Atom\ Bond} \rightarrow \Omega) \rightarrow \{\mathit{Vertex\ Atom\ Bond}\} \rightarrow \mathit{Nat}$
$\mathit{domCard} : (\mathit{Edge\ Atom\ Bond} \rightarrow \Omega) \rightarrow \{\mathit{Edge\ Atom\ Bond}\} \rightarrow \mathit{Nat}$
$\mathit{setExists}_1 : (\mathit{Molecule} \rightarrow \Omega) \rightarrow \{\mathit{Molecule}\} \rightarrow \Omega$
$(\mathit{subgraphs}\ 3) : \mathit{Molecule} \rightarrow \{\mathit{Molecule}\}$
$\mathit{msetExists}_2 : (\mathit{Vertex\ Atom\ Bond} \rightarrow \Omega) \rightarrow (\mathit{Vertex\ Atom\ Bond} \rightarrow \Omega)$
$\quad \rightarrow (\mathit{Vertex\ Atom\ Bond} \rightarrow \mathit{Nat}) \rightarrow \Omega$
$\wedge_2 : (\mathit{Charge} \rightarrow \Omega) \rightarrow (\mathit{Charge} \rightarrow \Omega) \rightarrow \mathit{Charge} \rightarrow \Omega$
$\wedge_2 : (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow \mathit{Atom} \rightarrow \Omega$
$\wedge_3 : (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow (\mathit{Atom} \rightarrow \Omega) \rightarrow \mathit{Atom} \rightarrow \Omega$
$\wedge_2 : (\mathit{Molecule} \rightarrow \Omega) \rightarrow (\mathit{Molecule} \rightarrow \Omega) \rightarrow \mathit{Molecule} \rightarrow \Omega$

$$\wedge_2 : (Edge\ Atom\ Bond \rightarrow \Omega) \rightarrow (Edge\ Atom\ Bond \rightarrow \Omega) \\ \rightarrow Edge\ Atom\ Bond \rightarrow \Omega.$$

Here is the predicate rewrite system.

$$
\begin{aligned}
&top \rightarrowtail (subgraphs\ 3) \circ (setExists_1\ (\wedge_2\ (vertices \circ top)\ (edges \circ top)))\\
&top \rightarrowtail (domCard\ top) \circ (> 0)\\
&top \rightarrowtail \wedge_2\ (connects \circ (msetExists_2\ top\ top))\ (edge \circ top)\\
&top \rightarrowtail vertex \circ projAtomType \circ top\\
&top \rightarrowtail (= Br)\\
&top \rightarrowtail (= C)\\
&top \rightarrowtail (= Cl)\\
&top \rightarrowtail (= F)\\
&top \rightarrowtail (= H)\\
&top \rightarrowtail (= I)\\
&top \rightarrowtail (= N)\\
&top \rightarrowtail (= O)\\
&top \rightarrowtail (= S)\\
&top \rightarrowtail (= 3)\\
&top \rightarrowtail (= 22)\\
&top \rightarrowtail (= 38)\\
&top \rightarrowtail (= 40)\\
&top \rightarrowtail (= 45)\\
&top \rightarrowtail (= 195)\\
&top \rightarrowtail (= 27)\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 2)\\
&top \rightarrowtail (= 3)\\
&top \rightarrowtail (= 7)\\
&(> 0) \rightarrowtail (> 1).
\end{aligned}
$$

For this setting, ALKEMY found the following definition for the function *active*.

$$
\begin{array}{l}
active\ m = \\
\quad if\ (subgraphs\ 3) \circ (setExists_1\ (\wedge_2 \\
\qquad (vertices \circ (domCard\ (vertex \circ projAtomType \circ (= 38))) \circ (> 0)) \\
\qquad (edges \circ (domCard\ (\wedge_2 \\
\qquad\quad (connects \circ (msetExists_2\ (vertex \circ projAtomType \circ (= 40))\ top)) \\
\qquad\quad (edge \circ (= 2)))) \circ (> 1))))\ m \\
\quad then\ \top \\
\quad else\ \bot .
\end{array}
$$

"If a molecule contains 3 atoms connected by bonds such that at least one atom has atom type 38, and at least two edges connect an atom having atom type 40 and also have bond type 2, then it is active; else it is not active".

Musk

The next illustration is the well-known Musk problem. The main interest in this problem is that it is a multiple-instance problem with which conventional learners have difficulty. Indeed, some authors have developed several special-purpose algorithms for solving the problem and showed that conventional learners performed badly. This problem is of interest because it fits naturally into the knowledge representation framework that has been developed in this book.

Briefly, the problem is to determine whether or not a molecule has a musk odour. The difficulty is that the molecules generally have many different conformations and, presumably, only one conformation is responsible for the activity. This problem is similar to the keys problem considered earlier with the conformations corresponding to the keys, but the conformations themselves are rather more complicated than keys. Each conformation is a tuple of 166 floating-point numbers, where 162 of these numbers represent the distance in angstroms from some origin in the conformation out along a radial line to the surface of the conformation and the other four numbers represent the position of a specific oxygen atom.

The experiment was carried out on the musk2 data set that contains 102 examples, of which 39 are musk and 63 are not. The 102 molecules have a total of 6598 conformations.

In the original Musk dataset, all 166 attributes are continuous values. This data was discretised as follows. For each attribute, the mean μ was calculated and the standard deviation σ of the values occurring in the data. Intervals are then built up centered on the mean, taking the width of each interval to be one standard deviation, and assigned integral labels to the intervals, so that interval 0 centred on the mean is $[\mu - \sigma/2, \mu + \sigma/2]$, interval 1 is $[\mu + \sigma/2, \mu + 3\sigma/2]$, interval -1 is $[\mu - 3\sigma/2, \mu - \sigma/2]$, and so on. In total, 13 intervals were chosen, labeled -6 through 6, as this covers most of the

distribution and gives adequate resolution. The outermost intervals, -6 and 6, were extended below and above respectively to cover any outlying points.

The projections are named $proj_1, \ldots, proj_{166}$. Thus a predicate such as $proj_{98} \circ (= 1)$ should be interpreted as meaning that along the radial line corresponding to the 98th component, the distance from the origin to the surface of the conformation lies between two particular values given in angstroms.

These considerations lead to the following declarations.

$$-6, -5, \ldots, 5, 6 : Distance$$
$$Conformation = Distance \times \cdots \times Distance$$
$$Molecule = \{Conformation\}.$$

Here the product type $Distance \times \cdots \times Distance$ contains 166 components. The function $musk$ to be learned has signature

$$musk : Molecule \rightarrow \Omega.$$

The background theory contains the following transformations.

$$(= -6) : Distance \rightarrow \Omega$$
$$\vdots$$
$$(= 6) : Distance \rightarrow \Omega$$
$$(\neq -6) : Distance \rightarrow \Omega$$
$$\vdots$$
$$(\neq 6) : Distance \rightarrow \Omega$$
$$proj_1 : Conformation \rightarrow Distance$$
$$\vdots$$
$$proj_{166} : Conformation \rightarrow Distance$$
$$setExists_1 : (Conformation \rightarrow \Omega) \rightarrow Molecule \rightarrow \Omega$$
$$\wedge_3 : (Conformation \rightarrow \Omega) \rightarrow (Conformation \rightarrow \Omega) \rightarrow (Conformation \rightarrow \Omega) \rightarrow Conformation \rightarrow \Omega.$$

Here is the predicate rewrite system.

$$top \rightarrowtail setExists_1 \, (\wedge_3 \; top \; top \; top)$$
$$top \rightarrowtail proj_1 \circ (= -6)$$
$$top \rightarrowtail proj_1 \circ (= -5)$$
$$\vdots$$
$$top \rightarrowtail proj_1 \circ (= 5)$$

$$top \rightarrowtail proj_1 \circ (= 6)$$
$$\vdots$$
$$top \rightarrowtail proj_{166} \circ (= -6)$$
$$top \rightarrowtail proj_{166} \circ (= -5)$$
$$\vdots$$
$$top \rightarrowtail proj_{166} \circ (= 5)$$
$$top \rightarrowtail proj_{166} \circ (= 6).$$

For this illustration, the default ALKEMY parameters were modified as follows. First, the *stump* parameter was set to true so that a decision-tree with a single split was requested in order to provide a suitable multiple-instance hypothesis. Second, because the predicate search space is so large, the *cutout* parameter was set to 30 000 so that the search terminated after 30 000 successive predicates were generated none of which improved the accuracy of the previous best predicate. With this setting, ALKEMY found the following definition for the function *musk*.

$$musk\ m =$$
$$\quad if\ setExists_1\ (\wedge_3\ (proj_{85} \circ (= 1))\ (proj_{93} \circ (= 0))\ (proj_{131} \circ (= -2)))\ m$$
$$\quad then\ \top$$
$$\quad else\ \bot.$$

Mutagenesis

The last illustration involves learning a theory to predict whether a chemical molecule is mutagenic or not. References are given in the Bibliographical Notes for background material on this important and interesting problem.

As before, an (undirected) graph was used to model a molecule. The type *Element* is the type of the (relevant) chemical elements.

$$Br, C, Cl, F, H, I, N, O, S : Element.$$

The following type synonyms are also made.

$$AtomType = Nat$$
$$Charge = Float$$
$$Atom = Element \times AtomType \times Charge$$
$$Bond = Nat$$
$$Molecule = Graph\ Atom\ Bond$$

The examples are the set of 188 regression-friendly molecules. For instance, the molecule d_1 from Example 3.5.3 in this set is mutagenic. The function *mutagenic* to be learned has signature

$$mutagenic : Molecule \rightarrow \Omega.$$

The background theory contains the following transformations.

$$(= Br) : Element \rightarrow \Omega$$

$$\vdots$$

$$(= S) : Element \rightarrow \Omega$$
$$(= 1) : AtomType \rightarrow \Omega$$
$$(= 3) : AtomType \rightarrow \Omega$$

$$\vdots$$

$$(= 230) : AtomType \rightarrow \Omega$$
$$(= 232) : AtomType \rightarrow \Omega$$
$$(\geq -0.781) : Charge \rightarrow \Omega$$
$$(\geq -0.424) : Charge \rightarrow \Omega$$
$$(\geq -0.067) : Charge \rightarrow \Omega$$
$$(\geq 0.290) : Charge \rightarrow \Omega$$
$$(\geq 0.647) : Charge \rightarrow \Omega$$
$$(\geq 1.004) : Charge \rightarrow \Omega$$
$$(= 1) : Bond \rightarrow \Omega$$
$$(= 2) : Bond \rightarrow \Omega$$
$$(= 3) : Bond \rightarrow \Omega$$
$$(= 4) : Bond \rightarrow \Omega$$
$$(= 5) : Bond \rightarrow \Omega$$
$$(= 7) : Bond \rightarrow \Omega$$
$$(> 0) : Nat \rightarrow \Omega$$
$$(> 1) : Nat \rightarrow \Omega$$
$$(> 2) : Nat \rightarrow \Omega$$
$$(> 3) : Nat \rightarrow \Omega$$
$$(> 4) : Nat \rightarrow \Omega$$
$$projElement : Atom \rightarrow Element$$
$$projAtomType : Atom \rightarrow AtomType$$
$$projCharge : Atom \rightarrow Charge$$
$$vertices : Molecule \rightarrow \{Vertex\ Atom\ Bond\}$$
$$edges : Molecule \rightarrow \{Edge\ Atom\ Bond\}$$
$$vertex : Vertex\ Atom\ Bond \rightarrow Atom$$
$$edge : Edge\ Atom\ Bond \rightarrow Bond$$

$$
\begin{aligned}
&connects : Edge\ Atom\ Bond \to (Vertex\ Atom\ Bond \to Nat)\\
&domCard : (Vertex\ Atom\ Bond \to \Omega) \to \{Vertex\ Atom\ Bond\} \to Nat\\
&domCard : (Edge\ Atom\ Bond \to \Omega) \to \{Edge\ Atom\ Bond\} \to Nat\\
&msetExists_2 : (Vertex\ Atom\ Bond \to \Omega) \to (Vertex\ Atom\ Bond \to \Omega)\\
&\qquad\qquad\qquad\qquad \to (Vertex\ Atom\ Bond \to Nat) \to \Omega\\
&\wedge_2 : (Charge \to \Omega) \to (Charge \to \Omega) \to Charge \to \Omega\\
&\wedge_2 : (Atom \to \Omega) \to (Atom \to \Omega) \to Atom \to \Omega\\
&\wedge_3 : (Atom \to \Omega) \to (Atom \to \Omega) \to (Atom \to \Omega) \to Atom \to \Omega\\
&\wedge_2 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to Molecule \to \Omega\\
&\wedge_3 : (Molecule \to \Omega) \to (Molecule \to \Omega) \to (Molecule \to \Omega)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to Molecule \to \Omega.
\end{aligned}
$$

There are many possible predicate rewrite systems for this illustration. Here is one that generates conditions on sets of atoms.

$$
\begin{aligned}
&top \rightarrowtail vertices \circ (domCard\ (vertex \circ top)) \circ (> 0)\\
&top \rightarrowtail \wedge_2\ (projAtomType \circ top)\ (projCharge \circ top)\\
&top \rightarrowtail (= 1)\\
&top \rightarrowtail (= 3)\\
&\qquad \vdots\\
&top \rightarrowtail (= 230)\\
&top \rightarrowtail (= 232)\\
&top \rightarrowtail (\geq -0.781)\\
&(\geq -0.781) \rightarrowtail (\geq -0.424)\\
&(\geq -0.424) \rightarrowtail (\geq -0.067)\\
&(\geq -0.067) \rightarrowtail (\geq 0.29)\\
&(\geq 0.29) \rightarrowtail (\geq 0.647)\\
&(\geq 0.647) \rightarrowtail (\geq 1.004)\\
&(> 0) \rightarrowtail (> 1).
\end{aligned}
$$

For this illustration, the default ALKEMY parameters were modified by turning post-pruning on and setting aside 15% of the examples as a validation set for post-pruning. For this setting, ALKEMY found the following definition.

$$
\begin{aligned}
&\mathit{mutagenic}\; m = \\
&\quad \mathit{if}\;\; \mathit{vertices} \circ (\mathit{domCard}\; (\mathit{vertex} \circ (\wedge_2 \\
&\qquad\qquad (\mathit{projAtomType} \circ (= 1))\; (\mathit{projCharge} \circ (\geq 0.290))))) \circ (> 0)\; m \\
&\quad \mathit{then}\; \bot \\
&\quad \mathit{else\; if}\;\; \mathit{vertices} \circ (\mathit{domCard}\; (\mathit{vertex} \circ (\wedge_2 \\
&\qquad\qquad\qquad (\mathit{projAtomType} \circ (= 50))\; (\mathit{projCharge} \circ \mathit{top})))) \circ (> 0)\; m \\
&\qquad \mathit{then}\; \bot \\
&\qquad \mathit{else}\; \top.
\end{aligned}
$$

"A molecule is mutagenic iff it does not have an atom of type 1 and charge ≥ 0.290, nor an atom of type 50."

Bibliographical Notes

For overviews of decision-tree learning, see [9], [61, Chap. 3] and [79, Chap. 18]. Widely-used decision-tree systems include CART [9], ID3 [73], and C4.5 [75]. The error-complexity post-pruning algorithm is described in [9]. First-order decision-trees are discussed in [3] and [4]. Related work on decision-trees is in [6], [44], and [89].

Feature construction, selection and extraction is discussed in [50] and [51]. The methodology of reducing a first-order learning problem to an attribute-value one is called propositionalisation in the inductive logic programming literature. In effect, propositionalisation is a methodology for constructing/selecting/extracting features. There is an extensive literature on propositionalisation including the recent works [45, 49] that contain references to much of the important earlier work.

For discussions of knowledge representation in machine learning, see [28] and [69]. The book chapter [28] also contains an excellent recent overview of inductive logic programming. Discussions of hypothesis language issues related to the approach of this chapter are in [13] and [42].

A higher-order approach to learning is advocated in [66]. Learning in a functional logic programming context is discussed in [25] and [26]. Learning with set-valued features is studied in [14]. For the application of some of the ideas of this book to evolutionary learning, see [40] and [41].

The book [20] contains several chapters that give first-order accounts of ideas that have been explored in this chapter and earlier in this book, including hypothesis language issues, decision trees, and metric methods.

The Tennis problem is described in [61, p.59] and [73]. The East–West challenge was originally posed in [47] and Fig. 6.4 is from that paper, although the actual figure used here appeared in [59]. Figure 6.4 is used with permission. Various alternative East–West data sets are available at [62]. The Bongard problem is from the book [5, p.229]. The Musk problem was

introduced in [18]. Discussion of the more general problem of pharmacophore discovery can be found in [18] and [27]. The Mutagenesis problem is discussed in [43] and [87].

While accuracies of the induced hypotheses in Sect. 6.2 are not indicated, the methods used there have been shown elsewhere [7, 8] to be competitive in this regard.

Proposition 6.1.1 appeared in [8]. Also the illustrations of Sect. 6.2 appeared in [8] in a similar form to their presentation here.

Exercises

6.1 Obtain the ALKEMY learning system and run the illustrations of Sect. 6.2.

6.2 For each of the predicate rewrite systems in Sect. 6.2, investigate whether it has a regularisation that is monotone or separable or descending or switchable or all four of these.

6.3 Choose a suitable application domain that you have already studied and apply ALKEMY to it. Enumerate the advantages and disadvantages of ALKEMY compared with the learning system that you used previously.

6.4 (Open problem) Find suitable new application domains and apply ALKEMY to them.

6.5 Let $T_1, \ldots, T_n$ be nullary type constructors such that there are finitely many constants of type T_i, for $i = 1, \ldots, n$. Let $proj_i : T_1 \times \cdots \times T_n \to T_i$ be the usual projection, for $i = 1, \ldots, n$. Consider the hypothesis language H for individuals of type $T_1 \times \cdots \times T_n$ given by the predicate rewrite system that contains the rewrite

$$top \rightarrowtail \wedge_n \ (proj_1 \circ top) \cdots (proj_n \circ top)$$

together with all rewrites of the form

$$top \rightarrowtail (= C),$$

where C is a constant of type T_i, for some $i \in \{1, \ldots, n\}$. Prove that the VC dimension of H is n.

6.6 (Open problem) Develop methods for calculating the VC dimension of hypothesis languages generated by predicate rewrite systems. Hence, or otherwise, determine bounds on the sample complexity for each of the illustrations in Sect. 6.2.

A. Appendix

A.1 Well-Founded Sets

This section contains some standard material on well-founded sets.

Recall that a *strict partial order* on a set A is a binary relation $<$ on A such that, for each $a, b, c \in A$, $a \not< a$ (irreflexivity), $a < b$ implies $b \not< a$ (asymmetry), and $a < b$ and $b < c$ implies $a < c$ (transitivity). In addition, a strict partial order is a *strict total order* if, for each $a, b \in A$, exactly one of $a = b$ or $a < b$ or $b < a$ holds.

If $<$ is a strict total order on a set A, then $<$ can be lifted to strict total order, also denoted by $<$, on the set of sequences of elements in A by $a_1 \ldots a_n < b_1 \ldots b_m$ if either
(i) $a_1 = b_1, \ldots, a_n = b_n$ and $n < m$, or
(ii) there exists j such that $1 \leq j \leq n$, $a_1 = b_1, \ldots, a_{j-1} = b_{j-1}$ and $a_j < b_j$.
The order $<$ on the sequences is called the *induced lexicographic* ordering.

Definition A.1.1. Suppose that $<$ is a strict partial order on a set A. Then $<$ is a *well-founded order* if there is no infinite sequence $a_1, a_2, \ldots$ such that $a_{i+1} < a_i$, for $i \in \mathbb{Z}^+$.

Thus a strict partial order is well-founded iff it does not admit an infinite strictly decreasing sequence. A set with a well-founded order is called a *well-founded set.* A characterisation of well-founded sets will be useful. For this, the concept of a minimal element will be needed.

Definition A.1.2. Let A be a set with a strict partial order $<$ and $X \subseteq A$. An element $a \in X$ is *minimal* for X if $x \not< a$, for all $x \in X$.

Proposition A.1.1. *Let A be a set with a strict partial order. Then A is a well-founded set iff every nonempty subset X of A has a minimal element (in X).*

Proof. Straightforward. □

Example A.1.1. If Σ any alphabet, then the set Σ^* of all strings over Σ with the substring relation $\prec$ (that is, $s_1 \prec s_2$ iff s_1 is a proper substring of s_2) is a well-founded set.

One can define a well-founded order on a product of well-founded sets.

Proposition A.1.2. *Let A_i be a set with a well-founded order $<_i$, for $i \in \{1, \ldots, n\}$. Define the relation $<$ on $A_1 \times \cdots \times A_n$ by $(a_1, \ldots, a_n) < (b_1, \ldots, b_n)$ if there exists $i \in \{1, \ldots, n\}$ such that $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$ and $a_i <_i b_i$. Then $<$ is a well-founded order on $A_1 \times \cdots \times A_n$.*

Proof. Straightforward. □

There is an induction principle for well-founded sets.

Proposition A.1.3. *Let A be a set with a well-founded order $<$. Let X be a subset of A satisfying the condition: for all $a \in A$, whenever $b \in X$, for all $b < a$, it follows that $a \in X$. Then $X = A$.*

Proof. Suppose that $X \neq A$. Thus $A \setminus X \neq \emptyset$ and so $A \setminus X$ has a minimal element a, say. Consider an element $b \in A$ such that $b < a$. By the minimality of a, it follows that $b \notin A \setminus X$ and thus $b \in X$. Since this is true for all $b < a$, it follows that $a \in X$, by the condition satisfied by X. This gives a contradiction and so $X = A$. □

The condition in Proposition A.1.3 satisfied by X implies that X must contain the minimal elements of A (since these have no predecessors).

There is also a principle of inductive construction on well-founded sets.

Proposition A.1.4. *Let A be a well-founded set and S a set. Then there exists a unique function $f : A \to S$ having arbitrary given values on the minimal elements of A and satisfying the condition that there is a rule that, for all $a \in A$, uniquely determines the value of $f(a)$ from the values $f(b)$, for $b < a$.*

Proof. Uniqueness is shown first. Suppose that there exist two distinct functions f and g having the same values on the minimal elements of A and satisfying the condition in the statement of the proposition. Let X be the set of elements of A on which f and g differ. Let a be a minimal element of X. Now a cannot be minimal in A because f and g agree on the minimal elements of A. Thus there exist elements $b \in A$ such that $b < a$. For such an element b, $f(b) = g(b)$, since $b \notin X$. By the condition satisfied by f and g, it follows that $f(a) = g(a)$, which gives a contradiction.

Next existence is demonstrated. Denote the set $\{b \in A \mid b \leq a\}$ by W_a. Let $X = \{a \in A \mid$ there exists a function f_a defined on W_a having the given values on the minimal elements of A in W_a and satisfying the uniqueness condition on $W_a\}$. I show by the induction principle of Proposition A.1.3 that $X = A$. Note that, if $a, b \in X$ and $b < a$, by the uniqueness part of the proof applied to W_b, it follows that $f_b(x) = f_a(x)$, for all $x \in W_b$. Suppose now that $a \in A$ and $b \in X$, for all $b < a$. By the previous remark, $a \in X$ – it suffices to define f_a by $f_a(b) = f_b(b)$, for all $b < a$, and let $f_a(a)$ be the value uniquely determined by the rule. By Proposition A.1.3, $X = A$. Now define f by $f(a) = f_a(a)$, for all $a \in A$. Clearly, f has the required properties. □

References

1. P.B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Kluwer Academic Publishers, second edition, 2002.
2. J.L. Bell. *Toposes and Local Set Theories.* Oxford Science Publications, 1988.
3. H. Blockeel and L. De Raedt. Top-down induction of logical decision trees. In K. Van Marcke and W. Daelemans, editors, *Proceedings of the Ninth Dutch Conference on Artificial Intelligence (NAIC'97)*, pages 333–342, Antwerpen, 1997.
4. H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101:285–297, 1998.
5. M. Bongard. *Pattern Recognition.* Spartan Books, 1970.
6. H. Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1194–1200. Morgan Kaufmann, 1995.
7. A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. Classification of individuals with complex structure. In P. Langley, editor, *Machine Learning: Proceedings of the Seventeenth International Conference (ICML2000)*, pages 81–88. Morgan Kaufmann, 2000.
8. A.F. Bowers, C. Giraud-Carrier, and J.W. Lloyd. A knowledge representation framework for inductive learning. `http://csl.anu.edu.au/~jwl`, 2001.
9. L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Chapman and Hall, 1984.
10. W. Buntine. Generalised subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36(2):149–176, 1988.
11. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
12. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
13. W.W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.
14. W.W. Cohen. Learning trees and rules with set-valued features. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 709–716, Menlo Park, CA, 1996. AAAI Press.
15. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines.* Cambridge University Press, 2000.
16. H.B. Curry and R. Feys. *Combinatory Logic.* North-Holland, 1958.
17. D. DeGroot and G. Lindstrom, editors. *Logic Programming: Relations, Functions and Equations.* Prentice-Hall, 1986.
18. T.G. Dietterich, R.H. Lathrop, and T. Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

19. A. Dovier, A. Policriti, and G. Rossi. Integrating lists, multisets, and sets in a logic programming framework. In F. Baader and K. Schulz, editors, *Proc. of FROCOS'96*, pages 213–229. Kluwer, 1996.
20. S. Džeroski and N. Lavrač, editors. *Relational Data Mining*. Springer, 2001.
21. Kerstin Eder. Implementing Escher on a graph reduction machine. In *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, pages 1–20, PLI'99, Paris, France, September 1999. K. Sagonas (Uppsala University) and P. Tarau (University of North Texas & BinNet Corporation, USA).
22. Kerstin Eder. A study of the operational behaviour of Escher and its implementation. In *Proceedings of the 8th International Workshop on Functional and Logic Programming*, pages 182–194, Institut IMAG - 46, avenue Felix Viallet, F-38031 Grenoble Cedex, France, June 1999. Rapport de Recherche RR 1021-I-Laboratoire LEIBNIZ.
23. Kerstin Eder. Yet another way of set-processing: The Escher style and its implementation. In *Proceedings of the Workshop on Declarative Programming with Sets (DPS'99)*, pages 37–50, I-43100 Parma, Italy, September 1999. Quaderni del Dipartimento di Matematica, n. 200, Universita degli Studi di Parma.
24. Kerstin I. Eder. *EMA: Implementing the Rewriting Computational Model of Escher*. PhD thesis, Department of Computer Science, University of Bristol, November 1998.
25. C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Learning functional logic classification concepts from databases. In *Proceedings of the 9th International Workshop on Functional and Logic Programming (WFLP 2000)*, pages 296–308. UPV University Press, Valencia, 2000/2039, 2000.
26. C. Ferri-Ramírez, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS 2001)*, pages 233–247. Springer, 2001. Lectures Notes in Computer Science 2024.
27. P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system PROGOL. *Machine Learning*, 30:241–270, 1998.
28. P. Flach and N. Lavrač. Learning in clausal logic: A perspective on inductive logic programming. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, pages 437–471. Springer, LNAI 2407, 2002. Essays in Honour of Robert A. Kowalski, Part I.
29. T. Gärtner, J.W. Lloyd, and P. Flach. Kernels for structured data. In S. Matwin and C. Sammut, editors, *Inductive Logic Programming, 12th International Conference, ILP2002*, Lecture Notes in Computer Science 2583, pages 66–83. Springer, 2003.
30. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
31. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
32. M. Hanus (ed.). Curry: An integrated functional logic language. `http://www.informatik.uni-kiel.de/~curry`.
33. D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California in Santa Cruz, Department of Computer Science, 1999.
34. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.
35. P.M. Hill and R.W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

36. R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
37. Home page of ILP workshops. `http://www.cs.york.ac.uk/ILP-events/`.
38. Home page of ILPnet2. `http://www.cs.bris.ac.uk/~ILPnet2/`.
39. S. Peyton Jones and J. Hughes (editors). Haskell98: A non-strict purely functional language. `http://haskell.org/`.
40. C.J. Kennedy and C. Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the Fourth International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA'99)*, pages 331–366. Springer, 1999.
41. C.J. Kennedy and C. Giraud-Carrier. Predicting chemical carcinogenesis using structural information only. In *Proceedings of the Third European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDDD'99)*, pages 360–365, 1999.
42. J.-U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*, chapter 16, pages 335–359. Academic Press, 1992.
43. R. King, S. Muggleton, S. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity in inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.
44. S. Kramer. Structural regression trees. In *Proceedings of 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 812–819. AAAI Press/MIT Press, 1996.
45. S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In S. Džeroski and N. Lavrač, editors, *Relational Data Mining*, pages 262–291. Springer, September 2001.
46. J. Lambek and P.J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.
47. J.B. Larson and R.S. Michalski. Inductive inference of VL decision rules. In *Workshop on Pattern-directed Inference Systems*, pages 23–27, 1977. Also in ACM SIGART Newsletter, 63, 1977.
48. J.-L. Lassez, M.J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, 1988.
49. N. Lavrač and P.A. Flach. An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic*, 2(4):458–494, October 2001.
50. H. Liu and H. Motoda, editors. *Feature Extraction, Construction and Selection: A Data Mining Perspective.* Kluwer, 1998.
51. H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer, 1998.
52. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second edition, 1987.
53. J.W. Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94*, pages 43–57. MIT Press, 1994.
54. J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3), March 1999.
55. J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. `http://csl.anu.edu.au/~jwl`, 2001.
56. J.W. Lloyd. Higher-order computational logic. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, pages 105–137. Springer, LNAI 2407, 2002. Essays in Honour of Robert A. Kowalski, Part I.

57. J.W. Lloyd. Predicate construction in higher-order logic. *Electronic Transactions on Artificial Intelligence*, 4(2000):21–51, Section B. `http://www.ep.liu.se/ej/etai/2000/009/`.
58. R.S. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20(2):111–161, 1983.
59. D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new east–west challenge. Oxford University Computing Laboratory, UK, 1994.
60. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
61. T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
62. Home page of MLnet. `http://www.mlnet.org/`.
63. S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.
64. S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Morgan Kaufmann, 1988.
65. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
66. S. Muggleton and C.D. Page. Beyond first-order learning: Inductive learning with higher order logic. Technical Report PRG-TR-13-94, Oxford University Computing Laboratory, 1994.
67. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
68. G. Nadathur and D.A. Miller. Higher-order logic programming. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Oxford University Press, 1998.
69. F. Neri and L. Saitta. Knowledge representation in machine learning. In F. Bergadano and L. De Raedt, editors, *Proceedings of the European Conference on Machine Learning*, pages 20–27. Springer, LNAI 784, 1994.
70. S.H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence 1228. Springer, 1997.
71. G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, 1970.
72. G.D. Plotkin. A further note on inductive generalization. In B. Beltzer and D. Michie, editors, *Machine Intelligence 6*, pages 101–124, 1971.
73. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
74. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.
75. J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
76. J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 135–151, 1970.
77. J. A. Robinson. *Logic: Form and Function*. Edinburgh University Press, 1979.
78. B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.
79. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, second edition, 2002.
80. C.A. Sammut. *Learning Concepts by Performing Experiments*. PhD thesis, University of South Wales, Australia, 1981.

81. C.A. Sammut. The origins of inductive logic programming. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming.* Jošef Stefan Institute, Ljubljana, 1993. Technical Report IJS-DP-6706.
82. C.A. Sammut and R.B. Banerji. Learning concepts by asking questions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 167–192. Morgan Kaufmann, 1986.
83. B. Schölkopf, C. Burges, and A. Smola, editors. *Advances in Kernel Methods: Support Vector Learning.* MIT press, 1998.
84. B. Schölkopf and A. Smola. *Learning with Kernels.* MIT Press, 2002.
85. E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.
86. A. Srinivasan, R.D. King, and S.H. Muggleton. The role of background knowledge: using a problem from chemistry to examine the performance of an ILP program. Technical Report PRG-TR-08-99, Oxford University Computing Laboratory, 1999.
87. A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of Fourth Inductive Logic Programming Workshop.* Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994. GMD-Studien Nr 237.
88. S. Vere. Induction of concepts in the predicate calculus. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 351–356, 1975.
89. L. Watanabe and L. Rendell. Learning structural decision trees from examples. In *Proceedings of 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 770 – 776. Morgan Kaufmann, 1991.
90. D.A. Wolfram. *The Clausal Theory of Types.* Cambridge University Press, 1993.

Notation

$()$, 38
$(\mathbb{Z}^+)^*$, 45
$=$, 38
$A_{\mathcal{E}}$, 208
$A_{\mathcal{P}}$, 208
$B_{\mathcal{P}}$, 209
$C : \alpha$, 38
$U_{t,\theta}$, 58
$V(t\ r)$, 94
$V_{t,\theta}$, 58
$\perp$, 38
$\exists x.t$, 43
$\forall x.t$, 43
κ_T, 116
$\langle p_0, p_1, \ldots, p_n \rangle$, 153
$\langle t_1, \ldots, t_1, \ldots, t_n, \ldots, t_n \rangle$, 110
$\longleftarrow$, 38
$\longleftrightarrow$, 38
$\longrightarrow$, 38
$\longrightarrow_\alpha$, 71
$\longrightarrow_\beta$, 71
$\longrightarrow_\eta$, 71
$\longrightarrow_{\beta\eta}$, 71
$\mathbb{F}$, 72
$\mathbb{N}$, 34
$\mathbb{Z}$, 45
$\mathbb{Z}^+$, 45
$\mathbf{R}$, 146
$\mathbf{S}$, 140
$\mathbf{S}_\alpha$, 140
$\mathcal{D}_\alpha$, 72
$\mathcal{O}(t)$, 46
$\mathcal{V}(t, \eta, I, \varphi)$, 74
$\mathfrak{B}$, 97
$\mathfrak{B}_\alpha$, 99
$\mathfrak{B}_m$, 100
$\mathfrak{C}$, 31
$\mathfrak{D}\mathfrak{S}_{\mathfrak{L}}$, 184
$\mathfrak{D}$, 84
$\mathfrak{D}_\alpha$, 85
$\mathfrak{D}_m$, 88
$\mathfrak{L}$, 40
$\mathfrak{N}$, 89
$\mathfrak{N}_\alpha$, 91
$\mathfrak{N}_m$, 92
$\mathfrak{P}$, 31
$\mathfrak{S}$, 32
$\mathfrak{S}^c$, 32
$\mathfrak{T}$, 31
$\mathfrak{V}$, 31
$\mathit{1}$, 31
Char, 34
$Constraints_{(s\ t)}$, 41
$Constraints_{(t_1, \ldots, t_n)}$, 41
Float, 34
Int, 34
List, 34
Nat, 34
Real, 34
String, 34
$Subst_{s \precsim t}$, 52
$arity(C)$, 39
$card(t)$, 114
domain, 35, 55
range, 35, 55
$supp(u)$, 101
$trace(s)$, 95
$\neg$, 38
$\overline{p}$, 147
$\rightarrow$, 31
$\sharp$, 38
$\overset{*}{\longrightarrow}_\alpha$, 71
$\overset{*}{\longrightarrow}_\beta$, 71
$\overset{*}{\longrightarrow}_\eta$, 71
$\overset{*}{\longrightarrow}_{\beta\eta}$, 71
$\theta\ |_t$, 55
$\times$, 31
$\top$, 38
$\varOmega$, 31
$\varPi$, 39

Σ, 39
ε, 45
φ, 106
ϱ_T, 106
$\vee$, 38
$\wedge$, 38
$\{\mu\}$, 44, 134
$\{t_1, \ldots, t_n\}$, 109
$\{\}$, 44
$\{x \mid t\}$, 44
o^t, 149
$p \Leftarrow q$, 158
$p \prec q$, 145
$p \preceq q$, 146
$p \rightarrowtail q$, 151
q^t, 149
$s < t$, 95
$s\, t$, 44
$s \approx t$, 55
$s \equiv t$, 93
$s \in t$, 44, 134
$s \precsim t$, 52
$t[s/r]_o$, 52
$t[s_1/r_1, \ldots, s_n/r_n]_{o_1,\ldots,o_n}$, 52
$t^{(i)}$, 65, 67
$t|_o$, 46

Index

about, 183
abstraction, 16, 40
accuracy
– of partition, 208
– of set of examples, 208
ALKEMY system, 210
α-conversion, 71
α-equivalent, 71
alphabet, 31
annotated term, 41
answer, 185
application, 17, 40
arity
– of data constructor, 24, 39
– of type constructor, 31
associated embedding, 121
associated Hilbert space, 121
associated mgu, 40
associated with, 39
asymmetry, 243
attribute-value language, 2
axiom, 76
– logical, 77
– proper, 77

background theory, 14, 143
– domain-specific, 15
– generic, 15
basic abstraction, 98
basic form of a normal term, 102
basic structure, 98
basic term, 23, 83, 97
basic tuple, 98
β-equivalent, 71
β-reduction, 71
$\beta\eta$-equivalent, 71
bias, 15
– language, 15
– search, 15
binary partition, 208
binding
– in term substitution, 55
– in type substitution, 35
body
– of predicate rewrite, 151
– of statement, 183
bound occurrence, 17, 48
bound variable, 49

cardinality of abstraction, 114
class, 14
classification problem, 14
closed
– term, 41
– type, 32
– type substitution, 35
composition of type substitutions, 35
comprehensible hypothesis, 15
computation, 185
computation step, 184
consistent theory, 76
constant, 31
constructor-separating, 124
curried signature, 18

data constructor, 24, 38
– default, 84
– full, 86
– nullary, 39
default data constructor, 84
default term, 25, 84
default value, 90
default-consistent, 123
definition, 184
denotation
– of constant, 73
– of term, 74
descending predicate rewrite system, 166
discrete kernel, 116
disjoint occurrences, 48
disjoint subterms, 48
distinguished singleton set, 31
domain, 72

domain-specific background theory, 15
domain-specific transformation, 139
dot product, 115

eligible subterm, 142
embedded conjunctively, 191
equivalent terms, 143
error-complexity post-pruning, 210
η-reduction, 71
example, 14
expected predicate, 174

feature space, 115
final predicate, 153
formula, 41
frame, 72
free occurrence, 17, 49
free variable, 39
full data constructor, 86
function, 24, 38
- positive definite, 115
- strictly positive definite, 130
- symmetric, 115
- target, 14

generalise, 15
generic background theory, 15
generic transformation, 139
goal, 185
grounding type substitution, 74

head
- of predicate rewrite, 151
- of statement, 183
hypothesis, 12
hypothesis language, 12, 15

idempotent term substitution, 56
identity substitution
- for term, 55
- for type, 35
implication preorder, 158
individual, 14
induced lexicographic ordering, 243
induction hypothesis, 32
initial predicate, 153
inner product, 115
instance
- by term substitution, 56
- by type substitution, 35
intended interpretation, 76
interpretation, 73
- intended, 76
inverse of type substitution, 36
invertible type substitution, 36
irreflexivity, 243

juxtaposition, 44

kernel, 115
- discrete, 116
- product, 116
- separating, 122

λ-conversion, 71
language, 40
language bias, 15
length of predicate derivation, 153
logical axiom, 77
logical consequence, 76
LR predicate derivation, 164

majority class, 207
metric, 106
metric space, 106
mgu, 36
minimal element, 243
model, 76
monotone argument, 177
monotone position, 177
monotone predicate rewrite system, 159
more general than
- for type substitutions, 36
- for types, 35
most general unifier, 36

normal abstraction, 90
normal structure, 90
normal term, 89
normal tuple, 90
nullary data constructor, 39
nullary type constructor, 31

occurrence, 46
occurrence set, 46
occurrences
- disjoint, 48
outermost redex, 184

parameter, 20, 31
parametric polymorphism, 20
partition, 208
- binary, 208
permutation of parameters, 36
polymorphism, 20
- parametric, 20
positional type, 63
positive definite function, 115

post-pruning, 210
– error-complexity, 210
predicate, 38
– expected, 174
– regular, 146
– standard, 140
predicate derivation, 153
– LR, 164
– regular, 169
predicate derivation step, 152
predicate rewrite, 151
– body of, 151
– head of, 151
predicate rewrite system, 151
– descending, 166
– monotone, 159
– regular, 162
– regularisation of, 162
– separable, 172
– switchable, 170
– weakly monotone, 176
predicate subderivation, 155
prefix of standard predicate, 140
principle of induction
– for well-founded sets, 244
– on structure of basic terms, 98
– on structure of closed types, 33
– on structure of default terms, 84
– on structure of normal terms, 90
– on structure of terms, 42
– on structure of types, 32
principle of inductive construction, 244
product kernel, 116
program, 184
proof, 77
proof system, 76
proper axiom, 77
proper prefix of standard predicate, 140
proper subterm, 47
proper suffix of standard predicate, 140
pseudometric, 106
pseudometric space, 105

rank, 131
rectified term, 50
redex, 152, 184
– outermost, 184
– selected, 152
refinement, 208
refinement bound, 209
regression problem, 14
regular predicate, 146
– switchable, 169
regular predicate derivation, 169
regular predicate rewrite system, 162
regularisation
– of predicate rewrite system, 162
– of standard predicate, 147
relative type
– of free variable, 39
– of subterm, 50
replacing set of subterms by terms, 52
replacing subterm by term, 52
residual type, 63
rule of inference, 76

satisfiable in interpretation, 76
satisfies formula in interpretation, 76
scalar product, 115
schema, 188
– statement, 188
scope, 49
search bias, 15
selected redex, 152
selection rule, 184
separable predicate rewrite system, 172
separating kernel, 122
signature, 12, 14, 17, 38
skeleton, 177
source, 131
space
– metric, 106
– pseudometric, 105
standard predicate, 140
– monotone, 158
– regularisation of, 147
– weakly monotone, 176
standardised apart, 40
statement, 183
statement schema, 188
strict kernel, 130
strict partial order, 243
strict total order, 243
strictly positive definite function, 130
subterm, 47
– eligible, 142
– of term at occurrence, 46
– proper, 47
subterms
– disjoint, 48
suffix of standard predicate, 140
supervised learning, 14
support, 101
switchable predicate rewrite system, 170
switchable regular predicate, 169
symmetric function, 115

symmetric transformation, 143

target, 131
target function, 14
term, 39
– annotated, 41
– basic, 23, 83, 97
– closed, 41
– default, 25, 84
– normal, 89
– rectified, 50
term substitution, 55
– idempotent, 56
test data, 14
theorem, 77
theory, 41
– consistent, 76
trace, 95
training data, 12, 14
transformation, 27, 131
– domain-specific, 139
– generic, 139
– symmetric, 143
transitivity, 243
transpose
– of a subterm, 149
– of an occurrence, 149
tuple, 17, 40
type, 31
– closed, 32
– of term, 39
– relative, 39, 50
type constructor, 17, 31
– nullary, 31
type substitution, 35
– closed, 35
– grounding, 74
– inverse of, 36
– invertible, 36
type theory, 1
type-equivalent, 55
type-weaker, 52

unifier, 36
unsupervised learning, 14
untyped λ-calculus, 17

valid, 76
valid in interpretation, 76
variable, 31
– bound, 49
– free, 39
variable assignment, 74
variant, 37

well-formed expression, 45
well-founded order, 243
well-founded set, 243

Cognitive Technologies

Managing Editors: D.M. Gabbay J. Siekmann

Editorial Board: A. Bundy J.G. Carbonell
M. Pinkal H. Uszkoreit M. Veloso W. Wahlster
M. J. Wooldridge

Advisory Board:
Luigia Carlucci Aiello
Franz Baader
Wolfgang Bibel
Leonard Bolc
Craig Boutilier
Ron Brachman
Bruce G. Buchanan
Luis Farinas del Cerro
Anthony Cohn
Koichi Furukawa
Georg Gottlob
Patrick J. Hayes
James A. Hendler
Anthony Jameson
Nick Jennings
Aravind K. Joshi
Hans Kamp
Martin Kay
Hiroaki Kitano
Robert Kowalski
Sarit Kraus
Kurt Van Lehn
Maurizio Lenzerini
Hector Levesque
John Lloyd
Alan Mackworth
Mark Maybury
Tom Mitchell
Johanna D. Moore
Stephen H. Muggleton
Bernhard Nebel
Sharon Oviatt
Luis Pereira
Lu Ruqian
Stuart Russell
Erik Sandewall
Luc Steels
Oliviero Stock
Peter Stone
Gerhard Strube
Katia Sycara
Milind Tambe
Hidehiko Tanaka
Sebastian Thrun
Junichi Tsujii
Andrei Voronkov
Toby Walsh
Bonnie Webber

Be the first to know

with the new online notification service

Springer Alert

You decide how we keep you up to date on new publications:

- Select a specialist field within a subject area
- Take your pick from various information formats
- Choose how often you'd like to be informed

And receive customised information to suit your needs

Register now

http://www.springer.de/alert

and then you are one click away from a world of computer science information!

Come and visit Springer's Computer Science Online Library

http://www.springer.de/comp

Springer

008939_001x_1c