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*

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.

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:

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

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

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

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

$$opens : Bunch \to \Omega.$$

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

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

The problem can now be stated more precisely. It is to find a definition for the function $\mathit{opens} : \mathit{Bunch} \to \Omega$ 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.

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

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

$$(\mathit{projMake}\ k) = \mathit{Abloy}.$$

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).

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

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; $\alpha \rightarrow \beta$ 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 $\alpha \rightarrow \beta$ 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 $Bunch \to \Omega$. 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 $\alpha \to \beta$; an expression of the form $(s\ t)$, where s is a term of type $\alpha \to \beta$ and t is a term of type 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

$[] : List\ Int$,
$\sharp : Int \to (List\ Int \to List\ Int)$,
$p : List\ Int \to \Omega$,
$q : List\ Int \to \Omega$, and
$\wedge : \Omega \to (\Omega \to \Omega)$.

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 $\to$ in type declarations are in order. It may seem more natural to use the signature

$$Int \times List\ Int \to List\ Int$$

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 $Int \to (List\ Int \to List\ Int)$ 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 $List\ Int \to List\ Int$. 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

$List\ Int \to List\ Int$, $((\sharp\ x)\ y)$ 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 $\Omega \to \Omega$, 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

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

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

$$(Int \to \Omega) \to \Omega$$

and with the intended meaning that Σ maps a predicate of type $Int \to \Omega$ 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 $r : Int \to \Omega$ 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

$$(Int \to \Omega) \to \Omega$$

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 $p : \mathit{List}\ a \to \Omega$ and $q : \mathit{List}\ \mathit{Int} \to \Omega$ be predicates. Then $(p\ x)$ is a term of type Ω in which the free variable x has relative type $\mathit{List}\ a$, $(q\ x)$ is a term of type Ω in which the free variable x has relative type $\mathit{List}\ \mathit{Int}$, $\lambda x.(p\ x)$ is a term of type $\mathit{List}\ a \to \Omega$, $\exists x.(p\ x)$ 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 $\mathit{List}\ \mathit{Int}$, and $(p\ x) \wedge (q\ x)$ is a term of type Ω in which the free variable x has relative type $\mathit{List}\ \mathit{Int}$.

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

$$\mathit{concat} : \mathit{List}\ a \to (\mathit{List}\ a \to \mathit{List}\ a)$$

defined by

$$\mathit{concat}\ [\,]\ z = z$$
$$\mathit{concat}\ (x \sharp y)\ z = x \sharp (\mathit{concat}\ y\ z).$$

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

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

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

Example 1.4.7. The function

$$\mathit{length} : \mathit{List}\ a \to \mathit{Int}$$

$$\mathit{length}\ [\,] = 0$$
$$\mathit{length}\ (x \sharp y) = 1 + \mathit{length}\ y$$

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

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

$$append\ (u, v, w)\ =\ (u = []\ \wedge\ v = w) \vee \exists r.\exists x.\exists y.(u = r\ \sharp\ x\ \wedge\ w = r\ \sharp\ y\ \wedge\ append\ (x, v, y))$$

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

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

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, [], ♯, *Null*, and *Node*. In general, data constructors have signatures of the form

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

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,

$$\mathit{Node}\ (\mathit{Node}\ \mathit{Null}\ 21\ \mathit{Null})\ 42\ (\mathit{Node}\ \mathit{Null}\ 73\ \mathit{Null})$$

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

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

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

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

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 (finite) sets, multisets, and so on. The term s_0 is a *default term*, $\perp$ 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

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

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

$$(= \mathit{Abloy}) : \mathit{Make} \to \Omega$$

defined by

$$((= \mathit{Abloy})\ x) = x = \mathit{Abloy}.$$

Thus $((= \mathit{Abloy})\ x) = \top$ iff $x = \mathit{Abloy}$. More generally, given a constant $C : \alpha$, there corresponds a predicate $(= C) : \alpha \to \Omega$ 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

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

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 $\mathit{projMake} \circ (= \mathit{Abloy})$ that has type $\mathit{Key} \to \Omega$. If k is a key, then $((\mathit{projMake} \circ (= \mathit{Abloy}))\ k) = \top$ iff the first component of k is *Abloy*.

Next consider the connective $\wedge : \Omega \to (\Omega \to \Omega)$ 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

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

defined by

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

Using $\wedge_2$, one can form the predicate

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

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

$$\mathit{setExists}_1 : (a \to \Omega) \to \{a\} \to \Omega$$

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

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