LOGIC AND DECLARATIVE LANGUAGE

MICHAEL DOWNWARD

TAYLOR & FRANCIS 1798 – 1998

Logic and Declarative Language

MICHAEL DOWNWARD
Thames Cancer Registry, London

UK Taylor & Francis Ltd, 1 Gunpowder Square, London EC4A 3DE
USA Taylor & Francis Inc., 1900 Frost Road, Suite 101, Bristol, PA 19007

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN 0–7484–0803–7 (HB)
ISBN 0–7484–0802–9 (PB)

Library of Congress Cataloging-in-Publication Data are available

Cover design by Youngs Design in Production

Typeset in Times 10/12pt by Graphicraft Typesetters Ltd, Hong Kong

Printed by T.J. International Ltd, Padstow, UK

Logic and Declarative Language

UK Taylor & Francis Ltd, 1 Gunpowder Square, London EC4A 3DE
USA Taylor & Francis Inc., 1900 Frost Road, Suite 101, Bristol, PA 19007

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN 0–7484–0803–7 (HB)
ISBN 0–7484–0802–9 (PB)

Library of Congress Cataloging-in-Publication Data are available

Cover design by Youngs Design in Production

Typeset in Times 10/12pt by Graphicraft Typesetters Ltd, Hong Kong

Printed by T.J. International Ltd, Padstow, UK

PART ONE

Logic without equality

Preface

In order to understand something of the problems facing the computer software industry we have also to understand something of its history. A great deal of computer programming is still carried out in the style of the third-generation programming languages introduced by John Backus in 1954. The basic problem of these languages is that they are derived from an earlier generation of assembler languages in which the programmer is directly responsible for allocating and manipulating memory locations in the underlying computer hardware. Languages such as Ada have added a great many details to the original formula translator produced by Backus, but the sad fact is that these changes might well have made the problem worse rather than better. In his 1977 Turing lecture the creator of FORTRAN commented, "Conventional programming languages have become fat and flabby," and noted ruefully that he bore some of the responsibility for this situation. These languages had been so successful in their early years that it became difficult to conceive any other way of implementing computer systems.

A computer program has to model a real-world situation and is best constructed by first specifying the nature of the system in some abstract form, then implementing the specification in a programming language. The problem with low-level languages such as Ada is that the gap between specification and implementation is uncomfortably large. Mistakes often arise as a result of an incorrect translation from specification to implementation. Just as important, programs written in machine-oriented languages inevitably require a great many lines of code because of their limited expressiveness. As a result, the probability of making mistakes while writing the code is greatly increased, even if the specification has been correctly understood. Programs written in these languages require comprehensive testing regimes, but even the best test routines will not catch every mistake.

It is particularly difficult to understand the meaning of machine-oriented computer programs because much of the program is concerned with movements between computer storage locations. Complete methodologies called formal methods such as the Vienna Definition Methodology (VDM) and the Z specification method were provided to define the translation from specification to machine-level code. Unfortunately these methodologies did little more than extend the problems of conventional programming languages to a higher specification level. The process of writing specifications at a high level then redrafting them at several lower levels resulted in very lengthy proofs. Some studies even suggested that formal methods of this kind might actually increase the number of mistakes as the complexity of development increased.

In fact programs written in conventional languages turned out to be more reliable than early observers expected. Traditional testing methods together with strict development regimes produce programs sufficiently reliable to control airliners and nuclear power stations. The real argument against conventional languages is not the mistakes that have to be removed from them, but the very low levels of productivity they allow. This is just one reason that explains the recent shift of interest away from procedural, machine-oriented languages to declarative languages. Conventional languages are procedural because they are used to instruct machines how to solve a problem by performing a certain sequence of actions. Declarative languages specify the problem in some form of logic, but leave the solution of the problem to a series of deductions in an underlying logic system. Programs written in declarative languages are sometimes described as "animated specifications", emphasising their similarity to specifications.

A large number of semantic modelling techniques have been invented with the intention of modelling real-world problems in some abstract form; they have much in common with declarative languages. Objects such as functions and relations found in declarative languages are much more likely to be of use in describing computer systems than statements involving machine storage locations. Specifications are statements in a form of logic and, in order to rationalise the vast number of different specification systems that have evolved, we need to understand their relationship with known logic systems. Progress has been greater than is generally realised and it seems certain that the intense effort in this direction will continue. As early as 1967 McCarthy wrote, "It is reasonable to hope that the relationship between computation and logic in the next century will be as fruitful as that between calculus and physics in the last." This vision of the future is all the more amazing in that it was written before the first papers describing relational databases, efficient logic programs, object-oriented programming, lazy functional languages, polymorphism, constructive logic and many other developments in logic and declarative language.

The task facing us in the twenty-first century is to explain all of the ad hoc developments of the past years in terms of a single coherent structure based in logic. It is already clear that classical logic alone will be inadequate and that other logics are required to act as specifications for computational structures. The interrelationship between intuitionistic logic and functional language that has been developed in recent years perhaps indicates the path we must follow. Other logics have corresponding relationships with declarative languages.

Logic has acquired a reputation for difficulty, perhaps because many of the approaches adopted have been more suitable for mathematicians than computer scientists. This book shows that the subject is not inherently difficult and that the connections between logic and declarative language are straightforward. Many exercises have been included in the hope they will lead to a much greater confidence in manual proofs, leading to a greater confidence in automated proofs.

ACKNOWLEDGEMENTS

A great debt of gratitude is owed to many friends and colleagues in and around the University of London, particularly to the academic staff at University College, Birkbeck College and King's College. My thanks also to an anonymous referee who patiently worked through the original typescript to produce valuable improvements. I am further indebted to the production staff at Taylor & Francis who guided the text through its late stages to final publication. Finally, at a very personal level, I would like to thank Margaret Dawson for her support during the long period it took to write this text.

Introduction

At the heart of the description of logic in this book there is a division between syntactic or proof theoretic reasoning and semantic or model theoretic reasoning. Syntactic reasoning proceeds through arguments based on the syntactic form of formulas and is based on the nature and order of symbols used in the formulas, whereas semantic reasoning depends on a meaning given to the formulas in some interpretation. Two forms of equality accompany this division: a syntactic equality compares strings of symbols and a semantic equality compares the values of terms. For example, the expression 6×7 is semantically equal to the number 42 because both terms denote the same value. This denotational form of equality is associated with meaning, representation or interpretation, as opposed to the syntactic form of equality, which is associated with the form or sense of an expression. Two expressions are syntactically equal when they contain the same symbols in the same order. Most people think that 6×7 is equal to 42 and are therefore implicitly using the semantic or denotational version of equality. As a result, it is usual to describe logic without the denotational form of equality simply as "logic without equality"; inclusion of denotational equality leads to "logic with equality".

Classical logic as described in Chapters 1 and 2 has evolved from the work of Frege during nineteenth century and, in common with other logics, uses predicates and terms as its major components. A predicate is either *true* or *false* in respect to some interpretation that is often expressed in the form of a relation. Such relations express information from which the truth of a predicate may be judged and are therefore concerned with semantics and meaning. A relation might consist of a collection of facts showing possible routes and distances between cities:

Route(*athens*,*rome*,*distance*(*athens*,*rome*))

Here a relation called *Route* has three arguments, the names of two cities and a function expressing the distance between them. Each of the three arguments is a term; the first two terms are simple constants and the third is a function that itself has two simple constant arguments. Since we know the distance in kilometres between these two cities, we know the value denoted by the application

$distance(athens,rome) = 1055$

and we might substitute this value in the *Route* fact above to give

$Route(athens,rome,1055)$

Term substitutions of this kind are only permitted in logic with equality because there is no concept of denotational equivalence in logic without equality.

Chapters 3 and 4 show that logic without equality can be animated to provide very expressive declarative languages that should properly be called relational languages, but are usually called logic languages. These languages are declarative in the sense that a programmer declares a collection of known and unknown terms within relations. An underlying reasoning system such as the SLD mechanism described in Chapter 3 then deduces possible values for the unknown terms. The important point is that a declarative language should arrive at its result without the need for explicit procedural directions for computing that result. Declarative languages are animated forms of specification in which computation is seen as automated deduction in an appropriate form of logic. Programming in logic first became a practical proposition when the Prolog language was introduced by Colmerauer and his group at the University of Marseilles in 1973. This fundamental work was rapidly developed by Kowalski, Van Emden and Warren at the University of Edinburgh, and efficient implementations of a language that became known as Edinburgh Prolog were widely available by 1980. One of the reasons for the success of logic languages such as Prolog is that they deliberately avoid equality because it leads to problems in logic language mechanisms.

Abstract types and object-oriented programming

Object-oriented programming has its philosophical basis in Birkhoff's work on universal algebras published during the 1930s, but the importance of this work in computer science was not recognised until the early 1970s. Separate groups led by Zilles at IBM, Guttag at the University of Southern California and Goguen at UCLA recognised the relevance of Birkhoff's work at almost the same time. The importance of this work grew from the simple observation that the concept of a type as it is used in computation is greater than that of a set. We tend, for example, to think of the set of integers $\{\ldots, -1, 0, 1, \ldots\}$ as a type, but these elements are of little use without a defined collection of operators that can be applied to the numbers. An abstract type encapsulates the operations and constants of a type into a single self-contained package called an abstract type or abstract data type (ADT). Different representations of the constants and operations in an abstract type are possible but each representation must behave in exactly the same way as the defining abstract type. Birkhoff's original work was restricted to homogeneous algebras, describing the behaviour of objects that contain elements of only one type. This work was later extended to heterogeneous systems with objects containing elements of different types. Goguen invented a much neater notation for what he called many-sorted abstract data types, making descriptions of heterogeneous objects almost as easy as that for homogeneous objects. An abstract type is a syntactic form in logic with equality and its objects act as representations or interpretations, providing a semantics for the type.

Abstract types define equalities between terms that allow fragments of large expressions to be substituted with equivalent values until the simplest possible form of an expression is obtained. This process of reduction to a simplest, so-called normal value is called term rewriting and is described in detail in Chapters 5 and 6. In addition to providing a method of specifying objects, Goguen and his colleagues also invented an object-oriented declarative language called OBJ that automated the process of term rewriting. Complex expressions presented to OBJ are rewritten according to a set of equations in a user-specified abstract type until the simplest possible canonical form is obtained. In this way OBJ acts as a rapid prototyping system because the specification is animated directly to reveal any flaws before any translation into a low-level conventional language takes place. OBJ is especially interesting because it is a programming language with a very clear and direct relationship to the first-order logic described in the early part of the text. Although it has a great deal in common with the functional languages described later in the text, it is important to note that functional languages are derived from a different starting-point. Fortunately, Goguen was able to show that the domain basis of functional languages can be seen as an extension of the object domains in a language such as OBJ. Many functional languages have an abstract type facility that allows operations to be encapsulated into a module that looks almost the same as an abstract type specification. As a result, these languages can act as animated object specifications in much the same way as OBJ.

Representations of abstract types are based on sets of elements called domains that may contain a special error element in each set. For example, in addition to the true and false constants of a Boolean domain there is a requirement for a third, error element. Abstract errors of this kind mean that a defined result is obtained when inappropriate arguments are submitted for evaluation. This requirement complicates the specifications a little, but in practice is handled without problems in the OBJ system.

Scott domains and functional languages

Scott domains are similar to the domains described above and were invented by Dana Scott to provide a semantics for a syntactic system called the lambda calculus. Domains of this kind act as a model for lambda calculus expressions in the same way that objects act as models for abstract types. Unlike simple term-rewriting systems, Scott domains give a special status to an operation that has failed to produce a defined result. Lifted domains in this theory include an additional bottom or null element that represents an undefined element in that domain. More important, functions in Scott's theory are defined in a way that sometimes enables them to produce a defined result even when applied to undefined arguments. Lifted domains of this kind are essential in simulating the behaviour of very large scale integrated (VLSI) circuits. At a very simple level we can picture a transistor in which +5 volts represents true and −5 volts represents false, but in order to get from +5 to −5 the voltage level has to pass through 0. In other words, the device has to pass though an undefined state on moving between defined states. This undefined state may be explicitly modelled by the null element of a lifted domain and as a consequence the transition become predictable.

Declarative languages based on Scott domains are usually called functional languages and have a great deal in common with term-rewriting systems based on object domains. McCarthy and Backus both realised the limitations of machine-oriented programming and both suggested functional languages as a solution to the problem. McCarthy introduced a language called Lisp that has been widely used over many years and has now become something of a cult language in certain areas. Some years later Backus introduced a language called FP that never became widely used, perhaps because so many other functional languages were by then available. Progress with functional language implementations continued, so that by 1990 Augustsson and Johnsson at Chalmers University in Sweden were able to announce a very efficient functional language called LML. Programs written in LML run almost as efficiently as those written in machine-oriented languages and have the additional advantage that they are more rapidly written and tested. The wide variety of functional languages available led a group of leading researcher workers in the field to define a "standard" functional language called Haskell. Although this language is now widely used in the research community and in some universities, it has not yet displaced the more familar functional languages. Chapter 8 of this book provides an outline of the Miranda functional language, probably the most popular system available at the moment.

Constructive logic

One of the most exciting developments in recent years has been the extension of type theory through constructive logic. Roughly stated, the early work of Curry and Howard establishes a one-to-one relationship between intuitionistic logic statements and type declarations. According to the Curry–Howard isomorphism, these apparently unconnected features of specifications and programming languages are two aspects of a single property. Constructive or intuitionistic logic differs from classical logic in that statements are always accompanied by their proof objects and these objects are in fact functional programs. As a result, a direct connection between logic and computation is established in a way never before possible. Using these techniques, the structures of computer programs are derived directly from logic statements. Existential quantifiers in constructive logic are identified with abstract types whereas universal quantifiers are identified with polymorphic programs. Consequently, two of the most important areas of modern computing are seen to arise naturally from constructive logic and new possibilities arise for the translation of specifications into programs.

Relational databases

Relational databases (RDBs) emerged from a paper published by Codd in 1970 and have since grown to become one of the most important software products in the computing industry. The success of relational databases has occurred in spite of, or perhaps because of, the remarkable simplicity of the relational model. A relational database consists of sets of records such as

```
{(123, 'Smith', 45.78),
(456, 'Gupta', 67.28),
(789, 'Patel', 87.36), ... }
```

and these sets are called tables in RDBs. Individual elements (attributes) within the records have to be entered in some arbitrary order, but the position of an element in the record is of no importance. Every column in the table is named and access to an element is through the column name, not through its position. A small number of relational database operations are then defined to operate on the tables of various databases, producing new sets of records as a result. Three of these operations are the familiar set union, set intersection and set difference operations; a further three or four are simple extensions of basic set theory operations. Relational database theory is little more than an extension of basic set theory, yet it is a sufficient basis for a huge industry.

One interesting feature of the relational model is that it is really a special case of a term-rewriting declarative language such as those described in the second half of this book. It is special in the sense that only set operations are applicable and only sets of records such as those described above are taken as arguments or operands. An interactive declarative language called Sequel (SQL) evolved as a user-friendly method of writing logic statements to describe relationships between known and unknown information in the database. The advantage of the relational model over earlier file-oriented processing languages such as COBOL follows the general advantage of declarative languages over procedural languages. A query is presented and satisfied by underlying software in an RDB without the explicit procedural instructions required in conventional languages.

Deductive databases

Logic languages such as Prolog are more expressive and capable of answering some queries that cannot be answered in the relational database model. On the other hand, the set-oriented processing approach adopted in RDBs is much more efficient than the exhaustive searches used in logic languages. A new set-oriented logic language called Datalog has been defined, combining many of the desirable features of both relational databases and logic languages. Developments of Datalog languages continued throughout the 1980s and eventually the Microelectronics and Computer Technology Corporation (MCC) produced a powerful extended Datalog system called LDL. This system has been implemented both on a parallel-processing machine and in a standard Unix environment. The increased use of parallel-processing machines should improve the speed of "data dredging", which these systems seem to do so well.

Functional databases

Databases are distinguished from file-oriented declarative languages by their ability to incorporate data changes in secondary storage incrementally as such changes are made. Persistent storage of this kind can be added to the basic features of a functional language to give functional databases that have some advantages over the longer-established relational databases. The advantages of functional databases over their relational counterparts flow from the ease with which specifications can be converted to implementations. The basic structures used in functional databases have much more in common with the semantic modelling techniques that can be used to describe real-world situations. Functional databases face the same efficiency problems that faced early relational databases, but because of the inherent advantages of the functional model, efforts to resolve these problems will continue.

Unified languages and parallel processing

Relations and functions are defined in logic and are used in conjunction with each other in describing theories in logic. Declarative languages, on the other hand, tend to be either exclusively relational or exclusively functional, and separate traditions have grown up in line with this division. This is unfortunate because many of the skills required to develop declarative language programs are derived from a declarative way of thinking rather than from a particular approach. Programmers have to learn to express what is required in an approriate form rather than provide explicit instructions for a machine. Much thought has been given to the idea of merging the two styles of declarative language into a single language that would then use relations or functions as appropriate. Chapter 10 includes a brief outline of the progress already made in this area.

Another major challenge facing computer scientists in the new century will be the production of software for computers with many processors. Parallel-processing machines are in fact already available, but our ability to write programs for them is sadly lacking. Hidden dependencies within conventional language programs prevent the easy migration of such programs to parallel-processing machines. Declarative language programs should not contain these dependencies, hence fragments of such programs may be distributed between processors in the computer. Research into the implementation of declarative languages on parallel processors continues at many centres around the world.

Contents