PEARSON

国外信息科学与技术优秀图书系列　计算机科学与技术

Haskell 函数式编程基础（第3版）

Haskell: The Craft of Functional Programming (Third Edition)

（英文版）

〔英〕Simon Thompson 著

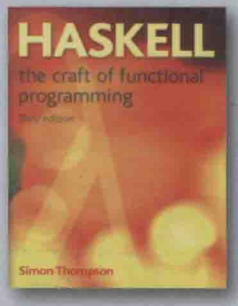
科学出版社

（TP-6354.0101）

本书简介

本书是一本非常优秀的Haskell函数式程序设计的入门书，各章依次介绍了函数式程序设计的基本概念、编译器和解释器、函数的各种定义方式、简单程序的构造、多态和高阶函数、诸如数组和列表的结构化数据、列表上的原始递归和推理、输入输出的控制处理、类型分类与检测方法、代数数据类型、抽象数据类型、惰性计算等内容。书中包含大量的实例和习题，注重程序测试、程序证明和问题求解，易读易学。全书循序渐进，从基本的函数式程序设计直至高级专题，让读者对Haskell的学习不断深入。

本书可作为计算机科学和其他相关学科的高年级本科生、研究生的教材，也可供对函数式程序设计感兴趣的程序员、软件工程师等参考学习。



国外信息科学与技术优秀图书系列

- 信息论基础（英文版）
- 经典与量子信息论（英文版）
- 信号处理与通信中的凸优化理论（英文版）
- 通信复杂性
- 认知无线电技术（第2版）
- 泊松点过程——成像、跟踪和感知
- 光电子器件——设计、建模与仿真
- 光波导模式——偏振、耦合与对称（英文版）
- 光放大原理与分析方法——增益介质中光的传播（英文版）
- 薄膜光学滤波器（第4版）
- 集成电路中的现代半导体器件（英文版）
- 功率半导体器件基础（英文版）
- 反馈控制系统（第5版，英文版）
- 矩阵数学通论（第2版，英文版）
- OpenCL编程指南（英文版）
- Haskell函数式编程基础（第3版，英文版）

PEARSON

For sale and distribution in the People's Republic of China exclusively (except Taiwan, Hong Kong SAR and Macau SAR).

仅限于中华人民共和国境内（不包括中国香港、澳门特别行政区和中国台湾地区）销售发行。

信息技术分社
联系电话：010-64009602
E-mail：it@mail.sciencep.com
销售分类建议：计算机语言、程序设计

PEARSON
www.pearson.com

www.sciencep.com

ISBN 978-7-03-037937-5

定 价：129.00 元

国外信息科学与技术
优秀图书系列

Haskell函数式编程基础（第3版）

Haskell: The Craft of Functional Programming (Third Edition)

（英文版）

PEARSON

科学出版社

国外信息科学与技术优秀图书系列

Haskell 函数式编程基础(英文版)

(第3版)

Haskell: The Craft of Functional Programming

(Third Edition)

〔英〕Simon Thompson 著

科学出版社

北京

图字：01-2013-1143 号

内 容 简 介

本书是一本非常优秀的 Haskell 函数式程序设计的入门书，各章依次介绍函数式程序设计的基本概念、编译器和解释器、函数的各种定义方式、简单程序的构造、多态和高阶函数、诸如数组和列表的结构化数据、列表上的原始递归和推理、输入输出的控制处理、类型分类与检测方法、代数数据类型、抽象数据类型、惰性计算等内容。书中包含大量的实例和习题，注重程序测试、程序证明和问题求解，易读易学。全书循序渐进，从基本的函数式程序设计直至高级专题，让读者对 Haskell 的学习不断深入。

本书可作为计算机科学和其他相关学科的高年级本科生、研究生的教材，也可供对函数式程序设计感兴趣的程序员、软件工程师等参考学习。

图书在版编目(CIP)数据

Haskell 函数式编程基础 = Haskell: the craft of functional programming : 第 3 版 : 英文 / (英) 汤普森 (Thompson, S.) 著. —北京：科学出版社，2013

(国外信息科学与技术优秀图书系列)

ISBN 978-7-03-037937-5

Ⅰ. ①H… Ⅱ. ①汤… Ⅲ. ①函数-程序设计-英文 Ⅳ. ①TP311.1

中国版本图书馆 CIP 数据核字(2013)第 136037 号

责任编辑：朱雪玲 / 责任校对：钟 洋 包志虹/ 责任印制：张 倩 / 封面设计：迷底书装

科 学 出 版 社出版

北京东黄城根北街16号

邮政编码：100717

http://www.sciencep.com

源海印刷有限责任公司印刷

科学出版社发行 各地新华书店经销

*

2013 年 7 月第 一 版 开本：787×1092 1/16

2013 年 7 月第一次印刷 印张：37 3/4

字数：600 000

定价：129.00 元

（如有印装质量问题，我社负责调换）

Preface

Computer technology changes with frightening speed; the fundamentals, however, remain remarkably static. The architecture of the standard computer is hardly changed from the machines which were built half a century ago, even though their size and power are incomparably different from those of today.

In programming, modern ideas like object-orientation have taken decades to move from the research environment into the commercial mainstream. In this light, a functional language like Haskell is a relative youngster, but one with a growing influence to play, particular as we move to multicore chips as the norm in all kinds of computing device. So, why is it a good idea to learn a functional programming language like Haskell?

- Functional languages are used to build secure, error-free, systems that are used in practice by thousands of people every day. For example, the *Xmonad* window manager for linux systems (`xmonad.org`) is written entirely in Haskell, and the *Cryptol* language (`www.cryptol.net`), used to design cryptographic systems in C and VHDL, is itself implemented using Haskell.
- Functional languages are general purpose programming languages, but also provide the ideal toolkit for building *Domain Specific Languages (DSLs)* which give users in a particular application area – such as hardware design or financial modelling – a language particularly suited to their needs.
- Functional languages provide a *laboratory* in which the crucial ideas of modern programming are presented in the clearest possible way. This accounts for their widespread use in teaching computing science and also for their influence on the design of other languages. A case in point is the design of generics in Java, which are directly modelled on polymorphism in Haskell.
- Functional languages help you *think about programming* in a different way from Java, C#, C++ and so forth: even if you are never going to write any large programs in Haskell, what you have learned will make you a better Java, C# or C++ programmer, because you can see a bigger space of possibilities for writing your code.
- Finally, it may well be that you will find yourself *working with a functional language* after all, even if it is not Haskell. Microsoft now provide the F# functional

language as a standard part of their Visual Studio suite, and this is in increasing use in a number of sectors, particularly finance. Erlang, the concurrent functional language, is used to provide scalable infrastructure for many web-based systems, including Facebook chat and other services from Amazon and Yahoo!

This book provides a tutorial introduction to functional programming in Haskell. The remainder of the preface begins with a brief explanation of functional programming, Haskell and GHCi, and continues by describing the intended audience for the book. To give a sense of how the book differs from others in the area we then summarise its distinctive points before giving a chapter-by-chapter summary of its contents. We then summarise how this edition differs from earlier ones, look at how the book can be read, and conclude by presenting a summary of the case studies it contains.

What is functional programming?

Functional programming offers a high-level view of programming, giving its users a variety of features which help them to build elegant yet powerful and general libraries of functions. Central to functional programming is the idea of a function, which computes a result that depends on the values of its inputs.

An example of the power and generality of the language is the `map` function, which is used to transform every element of a list of objects in a specified way. For example, `map` can be used to double all the numbers in a sequence of numbers or to invert the colours in each picture appearing in a list of pictures.

The elegance of functional programming is a consequence of the way that functions are defined: an equation is used to say what the value of a function is on an arbitrary input. A simple illustration is the function `addDouble` which adds two integers and doubles their sum. Its definition is

```
addDouble x y = 2*(x+y)
```

where `x` and `y` are the inputs and `2*(x+y)` is the result.

The model of functional programming is simple and clean: to work out the value of an expression like

```
3 + addDouble 4 5
```

the equations which define the functions involved in the expression are used, so

```
3 + addDouble 4 5
⇝ 3 + 2*(4+5)
⇝ 3 + 2*9
⇝ 3 + 18
⇝ 21
```

This is how a computer would work out the value of the expression, but it is also possible to do exactly the same calculation using pencil and paper, making transparent the implementation mechanism.

It is also possible to discuss how the programs behave in general. In the case of `addDouble` we can use the fact that `x+y` and `y+x` are equal for all numbers `x` and `y` to conclude that `addDouble x y` and `addDouble y x` are equal for all `x` and `y`. A *property* like this can be tested against random data, or indeed we can formally *prove* something like this from the definition of `addDouble`. Random testing and proof for properties like this are much more practical for Haskell than for traditional imperative and object-oriented (OO) languages.

Haskell and GHCi

This text uses the programming language Haskell, which has freely available compilers and interpreters for most types of computer system. Here we use the GHCi interpreter which provides an ideal platform for the learner, with its fast compile cycle, simple interface and free availability for Windows, Unix and Macintosh systems.

Haskell began life in the late 1980s as an intended standard language for lazy functional programming, and since then it has gone through various changes and modifications. This text is written in Haskell 2010, which is the latest standard for the language at the time of writing. The standard for Haskell is under yearly review, but it is likely that the parts of the language discussed here will be stable in future versions of the standard.

While the book covers most aspects of Haskell 2010, it is primarily a programming text rather than a language manual. Details of the language and its libraries as well as a wealth of other material about Haskell are available from the Haskell home page, `www.haskell.org`.

Who should read this book?

This text is intended as an introduction to functional programming for computer science and other students, principally at university level. It can be used by beginners to computer science, or more experienced students who are learning functional programming for the first time; either group will find the material to be new and challenging.

The book can also be used for self-study by programmers, software engineers and others interested in gaining a grounding in functional programming.

The text is intended to be self-contained, but some elementary knowledge of commands, files and so on is needed to use any of the implementations of Haskell. Some logical notation is introduced in the text; this is explained as it appears. In Chapter 20 it would help to have an understanding of the graphs of the `log`, n^2 and 2^n functions.

What is distinctive about the book?

Introductory programming texts will always have a lot in common, but each has its distinct ethos. These are the key aspects of *Haskell: The Craft of Functional Programming*:

- Haskell has a substantial library of built-in functions, particularly over lists, and we exploit this, encouraging readers to use these functions before seeing the details of their definitions. This allows readers to progress more quickly, and also accords

with practice: most real programs are built using substantial libraries of pre-existing code, and it is therefore valuable experience to work in this way from the start.

- From the start we introduce property-based testing with QuickCheck. This has shown itself to be a lightweight yet effective way of improving program quality, and examples throughout the book illustrate just that.
- The text gives a thorough treatment of reasoning about functional programs, beginning with reasoning about list-manipulating functions. These are chosen in preference to functions over the natural numbers for two reasons: the results one can prove for lists seem substantially more realistic, and also the structural induction principle for lists seems to be more acceptable to students. From the start, property-based testing and proof are compared and contrasted.
- The `Picture` case study is introduced in Chapter 1 and revisited throughout the text; this means that readers see different ways of programming the same function, and so get a chance to reflect on and compare different designs. The same interface for pictures is also implemented using browser-based graphics for realistic presentation.
- There is an emphasis on Haskell as a practical programming language, with an early introduction of modules, and the `do` notation for I/O. Other monadic programs are referred to later in the book.
- Types play a prominent role in the text. Every function or object defined has its type introduced at the same time as its definition. Not only does this provide a check that the definition has the type that its author intended, but also we view types as the single most important piece of documentation for a definition, since a function's type describes precisely how the function is to be used.
- A number of case studies are introduced in stages through the book: the picture example noted above, the game of Rock – Paper – Scissors, an interactive calculator program, regular expression processing, a coding and decoding system based on Huffman codes and a small queue simulation package. These are used to introduce various new ideas and also to show how existing techniques work together. There's an overview of what each case study covers on page xxi.
- A particular emphasis is laid on using Haskell for embedded domain-specific languages, with a chapter discussing this and giving a number of examples of monadic and combinator-based DSLs.
- Support materials on Haskell, including a substantial number of web links, are included in the concluding chapter. Various appendices contain other backup information including details of the availability of implementations, common GHCi errors and a comparison between functional, imperative and OO programming.
- Other support materials appear at `www.haskellcraft.com` and `www.pearsoned.co.uk/thompson`

Outline

The introduction in Chapter 1 covers the basic concepts of functional programming: functions and types, expressions and evaluation, definitions, proof and property-based testing with QuickCheck. Some of the more advanced ideas, such as higher-order

functions and polymorphism, are previewed here from the perspective of the example of pictures built from characters. A second implementation of pictures illustrates a discussion about domain-specific languages, which are the subject of Chapter 19. The picture examples is one of the running examples in the book, which we revisit a number of times to illustrate new concepts as they are introduced.

Chapter 2 looks at the practicalities of GHCi, the interactive version of the Glasgow Haskell Compiler. GHCi comes as a part of the Haskell platform, which is also introduced here. After looking at the basics of the module system, the standard prelude and the Haskell libraries, we look at a first exercise using an SVG implementation of the `Picture` type. These two chapters together cover the foundation on which to build a course on functional programming in Haskell.

Information on how to build simple programs over numbers, characters, strings and Booleans is contained in Chapter 3. The basic lessons are backed up with exercises, as is the case for all chapters from here on. With this basis, Chapter 4 steps back and examines the various strategies which can be used to define functions, such as auxiliary functions, local definitions and recursion. This chapter also introduces the simplest `data` types in the form of enumerated types. These types are used in the first discussion of the Rock – Paper – Scissors game which is another case study which we return to later in the book.

Structured data, in the form of tuples, lists and algebraic types come in Chapter 5. Algebraic types are used to represent products and sums, so giving records and variant records in other terminology. After introducing the idea of lists, programming over lists is performed using two resources: the list comprehension, which effectively gives the power of `map` and `filter`; and the first-order prelude and library functions.

Nearly all the list prelude functions are polymorphic, and so polymorphism is introduced at the start of Chapter 6, which also examines the list functions in the standard prelude, and then uses them in various extended examples. Only in Chapter 7 is primitive recursion over lists introduced, and a text processing case study provides a more substantial case study of how recursion is used in defining list functions.

Chapter 8 shows how simple terminal IO is handled in Haskell: to do this the `do` notation for writing programs of type `(IO a)` is introduced as an extension of Haskell's syntax, with the explanation of what underlies it being postponed to Chapter 18, where monads are covered. An interactive version of the Rock – Paper – Scissors game is used to illustrate IO in practice.

Chapter 9 introduces reasoning about list-manipulating programs, on the basis of a number of introductory sections giving the appropriate logical background. Guiding principles about how to build inductive proofs are presented, together with a more advanced section on building successful proofs from failed attempts. The chapter also describes the links between property-based testing and proof.

Higher-order functions are introduced in Chapters 10 and 11. First, functional arguments are examined, and it is shown that functional arguments allow the implementation of many of the 'patterns' of computation identified over lists at the start of the chapter. Chapter 11 covers functions as data, defined both as lambda-expressions and by partial application. These ideas are illustrated in Chapter 12 by revisiting a number of running examples, including pictures and the RPS game, as well as introducing new case studies of regular expression processing and index creation.

Type classes allow functions to be overloaded to mean different things at different types; Chapter 13 covers this topic as well as surveying the various classes built into Haskell, and exploring the way in which types are checked in Haskell. In general, type checking is a matter of resolving the various constraints put upon the possible type of the function by its definition.

Algebraic types like trees are the subject of Chapter 14, which covers all aspects of algebraic types from design and proof to their interaction with type classes, as well as introducing numerous examples of algebraic types in practice. These examples are consolidated in Chapter 15, which contains the case study of coding and decoding of information using a Huffman-style code. The foundations of the approach are outlined before the implementation of the case study. Modules are used to break the design into manageable parts, and the more advanced features of the Haskell module system are introduced at this point.

An abstract data type (ADT) provides access to an implementation through a restricted set of functions. Chapter 16 explores the ADT mechanism of Haskell and gives numerous examples of how it is used to implement queues, sets, relations and so forth, as well as giving the basics of a simulation case study.

Chapter 17 introduces lazy evaluation in Haskell which allows programmers a distinctive style incorporating backtracking and infinite data structures. As an example of backtracking there is a parsing case study, and infinite lists are used to give 'process style' programs as well as a random-number generator.

Haskell programs can perform input and output by means of the `IO` types, first introduced in Chapter 8. Chapter 18 revises this, and illustrates some larger-scale examples, including an interactive front-end to the calculator. The foundations of the `do` notation lie in monads, which can also be used to do action-based programming of a number of different flavours, some of which are examined in the second half of the chapter.

Domain–specific languages are one area where Haskell has been used very effectively. Chapter 19 explains what a DSL is, how a DSL can be embedded in a language like Haskell, and the distinction between shallow and deep embeddings. Example DSLs build on earlier case studies, as well as introducing new ones such as the generator language for QuickCheck, which neatly illustrates a monadic DSL.

The text continues with an examination in Chapter 20 of program behaviour, by which we mean the time taken for a program to compute its result, and the space used in that calculation. It also explains the basics of how to measure the run-time behaviour of Haskell programs. Chapter 21 concludes by surveying various applications and extensions of Haskell as well as looking at further directions for study. These are backed up with web and other references.

The appendices cover various background topics. The first examines links with functional and OO programming, and the second gives a glossary of commonly used terms in functional programming. The others include a summary of Haskell operators and GHCi errors, together with details of the various implementations of Haskell. The final appendix contains suggestions for larger-scale Haskell projects.

The Haskell code for all the examples in the book, as well as other background materials, can be downloaded as explained on `www.haskellcraft.com`.

What has changed from the second edition?

The third edition has seen changes throughout. Material, particularly by way of new examples, has been added to every chapter, and the order of presentation has changed in response to feedback on the previous edition. In detail the changes are these:

- *QuickCheck* is used throughout to test Haskell functions. Properties are developed right from the start, and QuickCheck is used to verify them – or indeed to show that properties can be erroneous too. HUnit is also introduced, but is used less intensively.
- QuickCheck is presented as complementary to *proof*, which has always been included in the book: QuickCheck can provide strong evidence for a property holding, while proof can establish its validity. Some custom generators are supplied in the code base for the book so that programmers can test their code before the details of how generators can be defined are discussed in Chapter 19.
- A number of *new examples* have been included, some in a single place and others running through a number of chapters. These include the Rock – Paper – Scissors (RPS) game, card games in general, an SVG rendering of Pictures and regular expression. There is a particular emphasis on using *functions as data*: they appear as strategies in RPS and as recognizers for regular expressions.
- One area where Haskell has been particularly successful is in providing the substrate for developing embedded *domain-specific languages* (DSLs) and Chapter 19 is devoted to this. It begins by explaining the reasons for developing DSLs, and then examines the difference between shallow and deep embeddings, looking at the examples of pictures and regular expressions. It concludes with a discussion of *monadic* DSLs, exemplified by naming in a pictures DSL and by the generators of QuickCheck.
- The text has been *reordered* so that some material comes earlier than it did previously.[1] Material on `data` types comes substantially earlier, with enumerated types coming into Section 4.3 and non-recursive types into Section 5.3. Programming for IO interactions is introduced in Chapter 8 to support the Rock – Paper – Scissors example: this treatment simply presents the `do` notation as the way that IO is programmed, and delays an explanation of the underlying mechanism to Chapter 18. Finally, local definitions first come into Section 4.2.
- The second edition used Hugs as its preferred implementation; in this edition we have moved to using *GHCi*, which comes as a part of the Haskell Platform. As well as introducing GHCi, we have added detailed discussions of how to leverage the best from Haskell libraries and packages through using Hackage, Cabal, Hoogle and Hayoo!

[1] It is intriguing that requests to move material forward outnumber those to delay it by a factor of more than ten to one: perhaps the ideal book introduces all its material in the first chapter, and uses the remaining twenty to discuss and expand on it?

- A *realistic* implementation of pictures – using the SVG / HTML5 capabilities of modern web browsers – has been added to the 'ASCII art' implementation of the second edition.
- The discussions in Chapter 20 on *performance* have been supplemented with material on how to measure the performance of real programs in GHC.
- A collection of *project suggestions* has been added as an appendix. Further support materials are available at the homepage `www.haskellcraft.com`, and solutions to exercises are available to bona fide instructors by application to the publishers.

What changed from the first edition to the second?

These changes were reported in detail in the preface to the second edition, but in summary the changes were these.

- The approach to defining functions over lists. To avoid the situation in which students try to define each new function over lists by recursion, we first introduced list comprehensions and list library functions, only introducing recursion after that. The jury is out over whether this effected the change it was meant to.
- The introduction of the `Pictures` case study as a running theme, giving visual feedback on programs, and also showing the role of higher-order, polymorphic functions in general-purpose Haskell list processing. An example I still like, but seen by some as making Haskell look lame.
- The edition was Haskell 98 compliant, and in particular used standard names for standard functions. It also contained an introduction to the Hugs interpreter, which was the implementation of choice for the book.
- Using the `do` notation, rather than the functional notation of `>>=` and `return`, for I/O programs and monadic programs in general.
- A more thorough treatment of Haskell typing and the mechanics of type checking in practice.
- The addition of some material on a problem-solving approach to getting started with programming, loosely based on Polyá's work.

How to read this book

This introduction to functional programming in Haskell is designed to be read from start to finish. New material comes in though the book, and is illustrated as it is introduced by a mixture of new examples and running case studies. Some parts of the text stand apart from the general flow, and can safely be omitted to build a shorter course:

Program proof The book emphasizes program proof, in a thread starting with Chapter 9, and followed up in Sections 11.6, 14.7, 16.10 and 17.9.

Program performance Similarly, Chapter 20 gives a self-contained treatment of program time and space behaviour.

The case studies in the book are designed to illustrate particular points and constructs as well as to give examples of larger programs than single function definitions.

Pictures This is used to show the utility of lists in modelling, as well as the value of the higher-order and polymorphic functions over lists, such as `zipWith`, `map` and `(++)`. Pictures are also used to illustrate shallow and deep embeddings of domain-specific languages in Chapter 19, with a variant of 'named' pictures giving an example of a monadic DSL.

Rock – Paper – Scissors In this example, we see `Move` as a first example of an enumerated type in Section 4.3, and strategies as an example of 'functions as data' in Section 8.1. It also provides an example of an IO interaction later in that chapter.

Calculator The calculator example begins in Section 14.2 with the `Expr` type, a recursive algebraic data type describing numerical expressions. Chapter 16 introduces the abstract type of `Store` used to model the values of the variables. In Section 17.5 we see how to parse text into numerical expressions and finally in Section 18.3 we give an interactive read-evaluate-print loop for calculation.

Library database When this is introduced in Section 5.7 it shows how to build programs over lists using list comprehensions, without using explicit recursion.

Supermarket billing This example comes in Section 6.7 and again illustrates the utility of the list library functions independently of explicit recursive definitions.

Text processing In text processing, Section 7.6, we present an example where explicit recursion over lists is necessary, in contrast to the previous two.

Regular expressions Regular expressions are first introduced as an example of 'functions as data', and then discussed again in Chapter 19 when a deep embedding is developed to contrast with the earlier treatment, which is a shallow embedding.

Huffman codes Chapter 15 is devoted to this multi-module application. The principal purpose of this is to show a larger example of programming in practice.

Other examples Other examples of simulation, relations and graphs are used to illustrate lazy evaluation in Chapter 17.

Acknowledgements

A book can only be improved by feedback, and I am very grateful to everyone who has contributed help and comments. Particular thanks are due to Thomas Schilling, whose advice on using cabal and hackage, as well as on the wider Haskell landscape, was absolutely invaluable. Eerke Boiten kindly proofread material on card games, and Olaf Chitil had a host of practical suggestions from his recent teaching experience. They and other colleagues from Kent and the USA contributed to lively discussions on what it means to be a DSL in Haskell, and the role of monads in DSLs.

Colleagues at Erlang Solutions in London were kind enough to provide an audience for the 'Haskell Evening Class' where some of this material was used; their questions and discussions helped to sharpen the presentation in many places.

Rufus Curnow of Addison-Wesley has supported this third edition from its inception almost to delivery; thanks very much to him for his hard work and patience, and also to Simon Lake who took over the task in the final stages. The anonymous referees were focused and constructive in their advice, both on my suggested changes and their own.

This book would not exist without all the efforts of those in the Haskell community, supporting first-class systems like GHC, and the burgeoning number of open source packages and applications that make Haskell such a pleasure to use: thanks very much to all of you!

Much appreciated sabbatical leave from the University of Kent has provided me with the time to work on this new edition; without this I am sure it would have taken many more years for the book to come to light.

Finally I thank my family for their support, understanding and encouragement while I was writing this: I dedicate the book to them.

Simon Thompson
Canterbury, December 2010

Publisher's Acknowledgements

The publisher would like to thank the following for their kind permission to reproduce their photographs:

Fotolia.com: Sean Gladwell / Fotolia 79, 178

Every effort has been made to trace the copyright holders and we apologise in advance for any unintentional omissions. We would be pleased to insert the appropriate acknowledgement in any subsequent edition of this publication.

Contents