真实世界的*Haskell*（影印版）

Real World

Haskell

O'REILLY®
東南大學出版社

Bryan O'Sullivan,
John Goerzen & Don Stewart 著

真实世界的Haskell(影印版)

Real World Haskell

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc.授权东南大学出版社出版

东南大学出版社

图书在版编目（CIP）数据

真实世界的Haskell：英文 /（美）沙利文（Sullivan, B.O'.），（美）戈尔（Goerzen, J.），（美）斯图尔特（Stewart, D.），著．—影印本．—南京：东南大学出版社，2010.1

书名原文：Real World Haskell

ISBN 978-7-5641-1925-6

Ⅰ．真… Ⅱ．①沙… ②戈… ③斯… Ⅲ．Haskell语言－程序设计－英文 Ⅳ．TP312

中国版本图书馆CIP数据核字（2009）第205748号

江苏省版权局著作权合同登记

图字：10-2009-241号

真实世界的Haskell（影印版）

出版发行：东南大学出版社

地　　址：南京四牌楼2号　　邮编：210096

出 版 人：江　汉

网　　址：http://press.seu.edu.cn

电子邮件：press@seu.edu.cn

印　　刷：扬中市印刷有限公司

开　　本：787毫米×980毫米　16开本

印　　张：44.5印张

字　　数：748千字

版　　次：2010年1月第1版

印　　次：2010年1月第1次印刷

书　　号：ISBN 978-7-5641-1925-6

印　　数：1~1600册

定　　价：88.00元（册）

本社图书若有印装质量问题，请直接与读者服务部联系。电话（传真）：025-83792328

To Cian, Ruairi, and Shannon, for the love and joy they bring.

—Bryan

For my wife, Terah, with thanks for all her love, encouragement, and support.

—John

To Suzie, for her love and support.

—Don

Preface

Have We Got a Deal for You!

Haskell is a deep language; we think learning it is a hugely rewarding experience. We will focus on three elements as we explain why. The first is *novelty*: we invite you to think about programming from a different and valuable perspective. The second is *power*: we'll show you how to create software that is short, fast, and safe. Lastly, we offer you a lot of *enjoyment*: the pleasure of applying beautiful programming techniques to solve real problems.

Novelty

Haskell is most likely quite different from any language you've ever used before. Compared to the usual set of concepts in a programmer's mental toolbox, functional programming offers us a profoundly different way to think about software.

In Haskell, we deemphasize code that modifies data. Instead, we focus on functions that take immutable values as input and produce new values as output. Given the same inputs, these functions always return the same results. This is a core idea behind functional programming.

Along with not modifying data, our Haskell functions usually don't talk to the external world; we call these functions *pure*. We make a strong distinction between pure code and the parts of our programs that read or write files, communicate over network connections, or make robot arms move. This makes it easier to organize, reason about, and test our programs.

We abandon some ideas that might seem fundamental, such as having a `for` loop built into the language. We have other, more flexible, ways to perform repetitive tasks.

Even the way in which we evaluate expressions is different in Haskell. We defer every computation until its result is actually needed—Haskell is a *lazy* language. Laziness is not merely a matter of moving work around, it profoundly affects how we write programs.

Power

Throughout this book, we will show you how Haskell's alternatives to the features of traditional languages are powerful and flexible and lead to reliable code. Haskell is positively crammed full of cutting-edge ideas about how to create great software.

Since pure code has no dealings with the outside world, and the data it works with is never modified, the kind of nasty surprise in which one piece of code invisibly corrupts data used by another is very rare. Whatever context we use a pure function in, the function will behave consistently.

Pure code is easier to test than code that deals with the outside world. When a function responds only to its visible inputs, we can easily state properties of its behavior that should always be true. We can automatically test that those properties hold for a huge body of random inputs, and when our tests pass, we move on. We still use traditional techniques to test code that must interact with files, networks, or exotic hardware. Since there is much less of this impure code than we would find in a traditional language, we gain much more assurance that our software is solid.

Lazy evaluation has some spooky effects. Let's say we want to find the k least-valued elements of an unsorted list. In a traditional language, the obvious approach would be to sort the list and take the first k elements, but this is expensive. For efficiency, we would instead write a special function that takes these values in one pass, and that would have to perform some moderately complex bookkeeping. In Haskell, the sort-then-take approach actually performs well: laziness ensures that the list will only be sorted enough to find the k minimal elements.

Better yet, our Haskell code that operates so efficiently is tiny and uses standard library functions:

```
-- file: ch00/KMinima.hs
-- lines beginning with "--" are comments.

minima k xs = take k (sort xs)
```

It can take a while to develop an intuitive feel for when lazy evaluation is important, but when we exploit it, the resulting code is often clean, brief, and efficient.

As the preceding example shows, an important aspect of Haskell's power lies in the compactness of the code we write. Compared to working in popular traditional languages, when we develop in Haskell we often write much less code, in substantially less time and with fewer bugs.

Enjoyment

We believe that it is easy to pick up the basics of Haskell programming and that you will be able to successfully write small programs within a matter of hours or days.

Since effective programming in Haskell differs greatly from other languages, you should expect that mastering both the language itself and functional programming techniques will require plenty of thought and practice.

Harking back to our own days of getting started with Haskell, the good news is that the fun begins early: it's simply an entertaining challenge to dig into a new language—in which so many commonplace ideas are different or missing—and to figure out how to write simple programs.

For us, the initial pleasure lasted as our experience grew and our understanding deepened. In other languages, it's difficult to see any connection between science and the nuts-and-bolts of programming. In Haskell, we have imported some ideas from abstract mathematics and put them to work. Even better, we find that not only are these ideas easy to pick up, but they also have a practical payoff in helping us to write more compact, reusable code.

Furthermore, we won't be putting any "brick walls" in your way. There are no especially difficult or gruesome techniques in this book that you must master in order to be able to program effectively.

That being said, Haskell is a rigorous language: it will make you perform more of your thinking up front. It can take a little while to adjust to debugging much of your code before you ever run it, in response to the compiler telling you that something about your program does not make sense. Even with years of experience, we remain astonished and pleased by how often our Haskell programs simply work on the first try, once we fix those compilation errors.

What to Expect from This Book

We started this project because a growing number of people are using Haskell to solve everyday problems. Because Haskell has its roots in academia, few of the Haskell books that currently exist focus on the problems and techniques of the typical programming that we're interested in.

With this book, we want to show you how to use functional programming and Haskell to solve realistic problems. We take a hands-on approach: every chapter contains dozens of code samples, and many contain complete applications. Here are a few examples of the libraries, techniques, and tools that we'll show you how to develop:

- Create an application that downloads podcast episodes from the Internet and stores its history in an SQL database.
- Test your code in an intuitive and powerful way. Describe properties that ought to be true, and then let the QuickCheck library generate test cases automatically.
- Take a grainy phone camera snapshot of a barcode and turn it into an identifier that you can use to query a library or bookseller's website.
- Write code that thrives on the Web. Exchange data with servers and clients written in other languages using JSON notation. Develop a concurrent link checker.

A Little Bit About You

What will you need to know before reading this book? We expect that you already know how to program, but if you've never used a functional language, that's fine.

No matter what your level of experience is, we tried to anticipate your needs; we go out of our way to explain new and potentially tricky ideas in depth, usually with examples and images to drive our points home.

As a new Haskell programmer, you'll inevitably start out writing quite a bit of code by hand for which you could have used a library function or programming technique, had you just known of its existence. We packed this book with information to help you get up to speed as quickly as possible.

Of course, there will always be a few bumps along the road. If you start out anticipating an occasional surprise or difficulty along with the fun stuff, you will have the best experience. Any rough patches you might hit won't last long.

As you become a more seasoned Haskell programmer, the way that you write code will change. Indeed, over the course of this book, the way that we present code will evolve, as we move from the basics of the language to increasingly powerful and productive features and techniques.

What to Expect from Haskell

Haskell is a general-purpose programming language. It was designed without any application niche in mind. Although it takes a strong stand on how programs should be written, it does not favor one problem domain over others.

While at its core, the language encourages a pure, lazy style of functional programming, this is the *default*, not the only option. Haskell also supports the more traditional models of procedural code and strict evaluation. Additionally, although the focus of the language is squarely on writing statically typed programs, it is possible (though rarely seen) to write Haskell code in a dynamically typed manner.

Compared to Traditional Static Languages

Languages that use simple static type systems have been the mainstay of the programming world for decades. Haskell is statically typed, but its notion of what types are for and what we can do with them is much more flexible and powerful than traditional languages. Types make a major contribution to the brevity, clarity, and efficiency of Haskell programs.

Although powerful, Haskell's type system is often also unobtrusive. If we omit explicit type information, a Haskell compiler will automatically infer the type of an expression or function. Compared to traditional static languages, to which we must spoon-feed large amounts of type information, the combination of power and inference in Haskell's type system significantly reduces the clutter and redundancy of our code.

Several of Haskell's other features combine to further increase the amount of work we can fit into a screenful of text. This brings improvements in development time and agility; we can create reliable code quickly and easily refactor it in response to changing requirements.

Sometimes, Haskell programs may run more slowly than similar programs written in C or C++. For most of the code we write, Haskell's large advantages in productivity and reliability outweigh any small performance disadvantage.

Multicore processors are now ubiquitous, but they remain notoriously difficult to program using traditional techniques. Haskell provides unique technologies to make multicore programming more tractable. It supports parallel programming, software transactional memory for reliable concurrency, and it scales to hundreds of thousands of concurrent threads.

Compared to Modern Dynamic Languages

Over the past decade, dynamically typed, interpreted languages have become increasingly popular. They offer substantial benefits in developer productivity. Although this often comes at the cost of a huge performance hit, for many programming tasks productivity trumps performance, or performance isn't a significant factor in any case.

Brevity is one area in which Haskell and dynamically typed languages perform similarly: in each case, we write much less code to solve a problem than in a traditional language. Programs are often around the same size in dynamically typed languages and Haskell.

When we consider runtime performance, Haskell almost always has a huge advantage. Code compiled by the Glasgow Haskell Compiler (GHC) is typically between 20 to 60 times faster than code run through a dynamic language's interpreter. GHC also provides an interpreter, so you can run scripts without compiling them.

Another big difference between dynamically typed languages and Haskell lies in their philosophies around types. A major reason for the popularity of dynamically typed

languages is that only rarely do we need to explicitly mention types. Through automatic type inference, Haskell offers the same advantage.

Beyond this surface similarity, the differences run deep. In a dynamically typed language, we can create constructs that are difficult to express in a statically typed language. However, the same is true in reverse: with a type system as powerful as Haskell's, we can structure a program in a way that would be unmanageable or infeasible in a dynamically typed language.

It's important to recognize that each of these approaches involves trade-offs. Very briefly put, the Haskell perspective emphasizes safety, while the dynamically typed outlook favors flexibility. If someone had already discovered one way of thinking about types that was always best, we imagine that everyone would know about it by now.

Of course, we, the authors, have our own opinions about which trade-offs are more beneficial. Two of us have years of experience programming in dynamically typed languages. We love working with them; we still use them every day; but usually, we prefer Haskell.

Haskell in Industry and Open Source

Here are just a few examples of large software systems that have been created in Haskell. Some of these are open source, while others are proprietary products:

- ASIC and FPGA design software (Lava, products from Bluespec, Inc.)
- Music composition software (Haskore)
- Compilers and compiler-related tools (most notably GHC)
- Distributed revision control (Darcs)
- Web middleware (HAppS, products from Galois, Inc.)

The following is a sample of some of the companies using Haskell in late 2008, taken from the Haskell wiki (*http://www.haskell.org/haskellwiki/Haskell_in_industry*):

ABN AMRO
: An international bank. It uses Haskell in investment banking, in order to measure the counterparty risk on portfolios of financial derivatives.

Anygma
: A startup company. It develops multimedia content creation tools using Haskell.

Amgen
: A biotech company. It creates mathematical models and other complex applications in Haskell.

Bluespec
: An ASIC and FPGA design software vendor. Its products are developed in Haskell, and the chip design languages that its products provide are influenced by Haskell.

Eaton
: Uses Haskell for the design and verification of hydraulic hybrid vehicle systems.

Compilation, Debugging, and Performance Analysis

For practical work, almost as important as a language itself is the ecosystem of libraries and tools around it. Haskell has a strong showing in this area.

The most widely used compiler, GHC, has been actively developed for over 15 years and provides a mature and stable set of features:

- Compiles to efficient native code on all major modern operating systems and CPU architectures
- Easy deployment of compiled binaries, unencumbered by licensing restrictions
- Code coverage analysis
- Detailed profiling of performance and memory usage
- Thorough documentation
- Massively scalable support for concurrent and multicore programming
- Interactive interpreter and debugger

Bundled and Third-Party Libraries

The GHC compiler ships with a collection of useful libraries. Here are a few of the common programming needs that these libraries address:

- File I/O and filesystem traversal and manipulation
- Network client and server programming
- Regular expressions and parsing
- Concurrent programming
- Automated testing
- Sound and graphics

The Hackage package database is the Haskell community's collection of open source libraries and applications. Most libraries published on Hackage are licensed under liberal terms that permit both commercial and open source use. Some of the areas covered by these open source libraries include the following:

- Interfaces to all major open source and commercial databases
- XML, HTML, and XQuery processing
- Network and web client and server development
- Desktop GUIs, including cross-platform toolkits
- Support for Unicode and other text encodings

A Brief Sketch of Haskell's History

The development of Haskell is rooted in mathematics and computer science research.

Prehistory

A few decades before modern computers were invented, the mathematician Alonzo Church developed a language called *lambda calculus*. He intended it as a tool for investigating the foundations of mathematics. The first person to realize the practical connection between programming and lambda calculus was John McCarthy, who created Lisp in 1958.

During the 1960s, computer scientists began to recognize and study the importance of lambda calculus. Peter Landin and Christopher Strachey developed ideas about the foundations of programming languages: how to reason about what they do (operational semantics) and how to understand what they mean (denotational semantics).

In the early 1970s, Robin Milner created a more rigorous functional programming language named *ML*. While ML was developed to help with automated proofs of mathematical theorems, it gained a following for more general computing tasks.

The 1970s also saw the emergence of lazy evaluation as a novel strategy. David Turner developed SASL and KRC, while Rod Burstall and John Darlington developed NPL and Hope. NPL, KRC, and ML influenced the development of several more languages in the 1980s, including Lazy ML, Clean, and Miranda.

Early Antiquity

By the late 1980s, the efforts of researchers working on lazy functional languages were scattered across more than a dozen languages. Concerned by this diffusion of effort, a number of researchers decided to form a committee to design a common language. After three years of work, the committee published the Haskell 1.0 specification in 1990. It named the language after Haskell Curry, an influential logician.

Many people are rightfully suspicious of "design by committee," but the output of the Haskell committee is a beautiful example of the best work a committee can do. They produced an elegant, considered language design and succeeded in unifying the fractured efforts of their research community. Of the thicket of lazy functional languages that existed in 1990, only Haskell is still actively used.

Since its publication in 1990, the Haskell language standard has seen five revisions, most recently in 1998. A number of Haskell implementations have been written, and several are still actively developed.

During the 1990s, Haskell served two main purposes. On one side, it gave language researchers a stable language in which to experiment with making lazy functional programs run efficiently and on the other side researchers explored how to construct programs using lazy functional techniques, and still others used it as a teaching language.

The Modern Era

While these basic explorations of the 1990s proceeded, Haskell remained firmly an academic affair. The informal slogan of those inside the community was to "avoid success at all costs." Few outsiders had heard of the language at all. Indeed, functional programming as a field was quite obscure.

During this time, the mainstream programming world experimented with relatively small tweaks, from programming in C, to C++, to Java. Meanwhile, on the fringes, programmers were beginning to tinker with new, more dynamic languages. Guido van Rossum designed Python; Larry Wall created Perl; and Yukihiro Matsumoto developed Ruby.

As these newer languages began to seep into wider use, they spread some crucial ideas. The first was that programmers are not merely capable of working in expressive languages; in fact, they flourish. The second was in part a byproduct of the rapid growth in raw computing power of that era: it's often smart to sacrifice some execution performance in exchange for a big increase in programmer productivity. Finally, several of these languages borrowed from functional programming.

Over the past half decade, Haskell has successfully escaped from academia, buoyed in part by the visibility of Python, Ruby, and even JavaScript. The language now has a vibrant and fast-growing culture of open source and commercial users, and researchers continue to use it to push the boundaries of performance and expressiveness.

Helpful Resources

As you work with Haskell, you're sure to have questions and want more information about things. The following paragraphs describe some Internet resources where you can look up information and interact with other Haskell programmers.

Reference Material

The Haskell Hierarchical Libraries reference
: Provides the documentation for the standard library that comes with your compiler. This is one of the most valuable online assets for Haskell programmers.

Haskell 98 Report
: Describes the Haskell 98 language standard.

GHC Users's Guide
: Contains detailed documentation on the extensions supported by GHC, as well as some GHC-specific features.

Hoogle and Hayoo
: Haskell API search engines. They can search for functions by name or type.

Applications and Libraries

If you're looking for a Haskell library to use for a particular task or an application written in Haskell, check out the following resources:

The Haskell community
: Maintains a central repository of open source Haskell libraries called Hackage (*http://hackage.haskell.org/*). It lets you search for software to download, or browse its collection by category.

The Haskell wiki (http://haskell.org/haskellwiki/Applications_and_libraries)
: Contains a section dedicated to information about particular Haskell libraries.

The Haskell Community

There are a number of ways you can get in touch with other Haskell programmers, in order to ask questions, learn what other people are talking about, and simply do some social networking with your peers:

- The first stop on your search for community resources should be the Haskell website (*http://www.haskell.org/*). This page contains the most current links to various communities and information, as well as a huge and actively maintained wiki.
- Haskellers use a number of mailing lists (*http://haskell.org/haskellwiki/Mailing_lists*) for topical discussions. Of these, the most generally interesting is named `haskell-cafe`. It has a relaxed, friendly atmosphere, where professionals and academics rub shoulders with casual hackers and beginners.
- For real-time chat, the Haskell IRC channel (*http://haskell.org/haskellwiki/IRC_channel*), named `#haskell`, is large and lively. Like `haskell-cafe`, the atmosphere stays friendly and helpful in spite of the huge number of concurrent users.
- There are many local user groups, meetups, academic workshops, and the like; there is a list of the known user groups and workshops (*http://haskell.org/haskellwiki/User_groups*).
- The Haskell Weekly News (*http://sequence.complete.org/*) is a very-nearly-weekly summary of activities in the Haskell community. You can find pointers to interesting mailing list discussions, new software releases, and similar things.
- The Haskell Communities and Activities Report (*http://haskell.org/communities/*) collects information about people that use Haskell and what they're doing with it. It's been running for years, so it provides a good way to peer into Haskell's past.

Conventions Used in This Book

The following typographical conventions are used in this book:

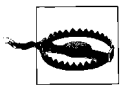
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Real World Haskell*, by Bryan O'Sullivan, John Goerzen, and Don Stewart. Copyright 2009 Bryan O'Sullivan, John Goerzen, and Donald Stewart, 978-0-596-51498-3."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596514983

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

This book would not exist without the Haskell community: an anarchic, hopeful cabal of artists, theoreticians and engineers, who for 20 years have worked to create a better, bug-free programming world. The people of the Haskell community are unique in their combination of friendliness and intellectual depth.

We wish to thank our editor, Mike Loukides, and the production team at O'Reilly for all of their advice and assistance.

Bryan

I had a great deal of fun working with John and Don. Their independence, good nature, and formidable talent made the writing process remarkably smooth.

Simon Peyton Jones took a chance on a college student who emailed him out of the blue in early 1994. Interning for him over that summer remains a highlight of my professional life. With his generosity, boundless energy, and drive to collaborate, he inspires the whole Haskell community.

My children, Cian and Ruairi, always stood ready to help me to unwind with wonderful, madcap, little-boy games.

Finally, of course, I owe a great debt to my wife, Shannon, for her love, wisdom, and support during the long gestation of this book.

John

I am so glad to be able to work with Bryan and Don on this project. The depth of their Haskell knowledge and experience is amazing. I enjoyed finally being able to have the three of us sit down in the same room—over a year after we started writing.

My 2-year-old Jacob, who decided that it would be fun to use a keyboard too and was always eager to have me take a break from the computer and help him make some fun typing noises on a 50-year-old Underwood typewriter.

Most importantly, I wouldn't have ever been involved in this project without the love, support, and encouragement from my wife, Terah.

Don

Before all else, I'd like to thank my amazing coconspirators, John and Bryan, for encouragment, advice, and motivation.

My colleagues at Galois, Inc., who daily wield Haskell in the real world, provided regular feedback and war stories and helped ensure a steady supply of espresso.

My Ph.D. supervisor, Manuel Chakravarty, and the PLS research group, who provided encouragement, vision, and energy and showed me that a rigorous, foundational approach to programming can make the impossible happen.

And, finally, thanks to Suzie, for her insight, patience, and love.

Thank You to Our Reviewers

We developed this book in the open, posting drafts of chapters to our website as we completed them. Readers then submitted feedback using a web application that we