A special issue of the *Journal of Functional Programming*

Contents

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

Preface

"*Some half dozen persons have written technically on combinatory logic, and most of these, including ourselves, have published something erroneous. Since some of our fellow sinners are among the most careful and competent logicians on the contemporary scene, we regard this as evidence that the subject is refractory. Thus fullness of exposition is necessary for accuracy; and excessive condensation would be false economy here, even more than it is ordinarily.*"

Haskell B. Curry and Robert Feys
in the Preface to *Combinatory Logic* [4], May 31 1956

In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages. This book describes the result of that committee's efforts: a purely functional programming language called Haskell, named after the logician Haskell B. Curry whose work provides the logical basis for much of ours.

Goals

The committee's primary goal was to design a language that satisfied these constraints:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be based on ideas that enjoy a wide consensus.
5. It should reduce unnecessary diversity in functional programming languages.

Haskell 98: language and libraries

The committee intended that Haskell would serve as a basis for future research in language design, and hoped that extensions or variants of the language would appear, incorporating experimental features. Haskell has indeed evolved continuously since its original publication. By the middle of 1997, there had been four iterations of the language design (the latest at that point being Haskell 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed; this stable language is the subject of this book, and is called *Haskell 98*.

Haskell 98 was conceived as a relatively minor tidy-up of Haskell 1.4, making some simplifications, and removing some pitfalls for the unwary. It is intended to be a "stable" language in sense the *implementors are committed to supporting Haskell 98 exactly as specified, for the foreseeable future*.

The original Haskell Report covered only the language, together with a standard library called the `Prelude`. By the time Haskell 98 was stabilised, it had become clear that many programs need access to a larger set of library functions (notably concerning input/output and simple interaction with the operating system). If these program were to be portable, a set of libraries would have to be standardised too. A separate effort was therefore begun by a distinct (but overlapping) committee to fix the Haskell 98 Libraries.

The Haskell 98 Language and Library Reports were published in February 1999.

Revising the Haskell 98 Reports

After a year or two, many typographical errors and infelicities had been spotted. I took on the role of gathering and acting on these corrections, with the following goals:

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

- Correct typographical errors.
- Clarify obscure passages.
- Resolve ambiguities.
- With reluctance, make small changes to make the overall language more consistent.

This task turned out to be much, much larger than I had anticipated. As Haskell becomes more widely used, the Report has been scrutinised by more and more people, and I have adopted hundreds of (mostly small) changes as a result of their feedback. The original committees ceased to exist when the original Haskell 98 Reports were published, so every change was instead proposed to the entire Haskell mailing list.

This book is the outcome of this process of refinement. It includes both the Haskell 98 Language Report and the Libraries Report, and constitutes the official specification of both. It is *not* a tutorial on programming in Haskell such as the "Gentle Introduction" [9], and some familiarity with functional languages is assumed.

The entire text of both Reports is available online (see 'Haskell Resources' on p. x).

Extensions to Haskell 98

Haskell continues to evolve, going well beyond Haskell 98. For example, at the time of writing there are Haskell implementations that support:

- **Syntactic sugar**, including:
  - pattern guards;
  - recursive do-notation;
  - lexically scoped type variables;
  - meta-programming facilities;
- **Type system innovations**, including:
  - multi-parameter type classes;
  - functional dependencies;
  - existential types;
  - local universal polymorphism and arbitrary rank-types;
- **Control extensions**, including:
  - monadic state;
  - exceptions;

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

– concurrency.

There is more besides. Haskell 98 does not impede these developments. Instead, it provides a stable point of reference, so that those who wish to write text books, or use Haskell for teaching, can do so in the knowledge that Haskell 98 will continue to exist.

Haskell Resources

The Haskell web site

```
http://haskell.org
```

gives access to many useful resources, including:

- Online versions of the language and library definitions, including a complete list of all the differences between Haskell 98 as published in February 1999 and this revised version.
- Tutorial material on Haskell.
- Details of the Haskell mailing list.
- Implementations of Haskell.
- Contributed Haskell tools and libraries.
- Applications of Haskell.

We welcome your comments, suggestions, and criticisms on the language or its presentation in the report, via the Haskell mailing list.

Building the language

Haskell was created, and continues to be sustained, by an active community of researchers and application programmers. Those who served on the Language and Library committees, in particular, devoted a huge amount of time and energy to the language. Here they are, with their affiliation(s).

Arvind (MIT)
Lennart Augustsson (Chalmers University)
Dave Barton (Mitre Corp)
Brian Boutel (Victoria University of Wellington)
Warren Burton (Simon Fraser University)
Jon Fairbairn (University of Cambridge)
Joseph Fasel (Los Alamos National Laboratory)
Andy Gordon (University of Cambridge)

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

Maria Guzman (Yale University)
Kevin Hammond (Uniiversity of Glasgow)
Ralf Hinze University of Bonn)
Paul Hudak [editor] (Yale University)
John Hughes [editor] (University of Glasgow; Chalmers University)
Thomas Johnsson (Chalmers University)
Mark Jones (Nottingham University)
Dick Kieburtz (Oregon Graduate Institute)
John Launchbury (University of Glasgow; Oregon Graduate Institute)
Erik Meijer (Utrecht University)
Rishiyur Nikhil (MIT)
John Peterson (Yale University)
Simon Peyton Jones [editor] (University of Glasgow; Microsoft Research Ltd)
Mike Reeve (Imperial College)
Alastair Reid (University of Glasgow)
Colin Runciman (University of York)
Philip Wadler [editor] (University of Glasgow)
David Wise (Indiana University)
Jonathan Young (Yale University)

Those marked [editor] served as the co-ordinating editor for one or more revisions of the language.

In addition, dozens of other people made helpful contributions, some small but many substantial. They are as follows: Kris Aerts, Hans Aberg, Sten Anderson, Richard Bird, Stephen Blott, Tom Blenko, Duke Briscoe, Paul Callaghan, Magnus Carlsson, Mark Carroll, Manuel Chakravarty, Franklin Chen, Olaf Chitil, Chris Clack, Guy Cousineau, Tony Davie, Craig Dickson, Chris Dornan, Laura Dutton, Chris Fasel, Pat Fasel, Sigbjorn Finne, Michael Fryers, Andy Gill, Mike Gunter, Cordy Hall, Mark Hall, Thomas Hallgren, Matt Harden, Klemens Hemm, Fergus Henderson, Dean Herington, Ralf Hinze, Bob Hiromoto, Nic Holt, Ian Holyer, Randy Hudson, Alexander Jacobson, Patrick Jansson, Robert Jeschofnik, Orjan Johansen, Simon B. Jones, Stef Joosten, Mike Joy, Stefan Kahrs, Antti-Juhani Kaijanaho, Jerzy Karczmarczuk, Wolfram Karl, Kent Karlsson, Richard Kelsey, Siau-Cheng Khoo, Amir Kishon, Feliks Kluzniak, Jan Kort, Marcin Kowalczyk, Jose Labra, Jeff Lewis, Mark Lillibridge, Bjorn Lisper, Sandra Loosemore, Pablo Lopez, Olaf Lubeck, Ian Lynagh, Christian Maeder, Ketil Malde, Simon Marlow, Michael Marte, Jim Mattson, John Meacham, Sergey Mechveliani, Erik Meijer, Gary Memovich, Randy Michelsen, Rick Mohr, Andy Moran, Graeme Moss, Arthur Norman, Nick North, Chris Okasaki, Bjarte M. Østvold, Paul Otto, Sven Panne, Dave Parrott, Ross Patterson, Larne Pekowsky, Rinus Plasmeijer, Ian Poole, Stephen Price, John Robson, Andreas Rossberg, George Russell, Patrick Sansom, Felix Schroeter, Julian Seward, Nimish Shah, Christian Sievers, Libor Skarvada, Jan Skibinski, Lauren Smith, Raman Sundaresh, Ken Takusagawa, Satish Thatte, Simon Thompson, Tom Thomson, Tommy Thorn, Dylan Thurston, Mike Thyer, Mark Tullsen, David Tweed, Pradeep Varma, Malcolm Wallace, Keith Wansbrough, Tony Warnock, Michael Webber, Carl Witty, Stuart Wray, and Bonnie Yantis.

Finally, aside from the important foundational work laid by Church, Rosser, Curry, and others on the lambda calculus, we wish to acknowledge the influence of many noteworthy programming

languages developed over the years. Although it is difficult to pinpoint the origin of many ideas, we particularly wish to acknowledge the influence of Lisp (and its modern-day incarnations Common Lisp and Scheme); Landin's ISWIM; APL; Backus's FP [1]; ML and Standard ML; Hope and Hope$^+$; Clean; Id; Gofer; Sisal; and Turner's series of languages culminating in Miranda.[1] Without these forerunners Haskell would not have been possible.

Simon Peyton Jones
Cambridge, November 2002

[1]Miranda is a trademark of Research Software Ltd.

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

Part I

The Haskell 98 Language

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

JFP **13** (1): i–xii, 1–6, January 2003.
DOI: 10.1017/S0956796803000315 Printed in the United Kingdom

Chapter 1

Introduction

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages.

This book defines the syntax for Haskell programs and an informal abstract semantics for the meaning of such programs. We leave as implementation dependent the ways in which Haskell programs are to be manipulated, interpreted, compiled, etc. This includes such issues as the nature of programming environments and the error messages returned for undefined programs (i.e. programs that formally evaluate to $\perp$).

1.1 Program Structure

In this section, we describe the abstract syntactic and semantic structure of Haskell, as well as how it relates to the organization of the rest of the report.

1. At the topmost level a Haskell program is a set of *modules*, described in Chapter 5. Modules provide a way to control namespaces and to re-use software in large programs.
2. The top level of a module consists of a collection of *declarations*, of which there are several kinds, all described in Chapter 4. Declarations define things such as ordinary values, datatypes, type classes, and fixity information.

3. At the next lower level are *expressions*, described in Chapter 3. An expression denotes a *value* and has a *static type*; expressions are at the heart of Haskell programming "in the small."

4. At the bottom level is Haskell's *lexical structure*, defined in Chapter 2. The lexical structure captures the concrete representation of Haskell programs in text files.

This book proceeds bottom-up with respect to Haskell's syntactic structure.

The chapters not mentioned above are Chapter 6, which describes the standard built-in datatypes and classes in Haskell, and Chapter 7, which discusses the I/O facility in Haskell (i.e. how Haskell programs communicate with the outside world). Also, there are several chapters describing the Prelude, the concrete syntax, literate programming, the specification of derived instances, and pragmas supported by most Haskell compilers.

Examples of Haskell program fragments in running text are given in typewriter font:

```
let x = 1
    z = x+y
in  z+1
```

"Holes" in program fragments representing arbitrary pieces of Haskell code are written in italics, as in `if` e_1 `then` e_2 `else` e_3. Generally, the italicized names are mnemonic, such as e for expressions, d for declarations, t for types, etc.

1.2 The Haskell Kernel

Haskell has adopted many of the convenient syntactic structures that have become popular in functional programming. In this Report, the meaning of such syntactic sugar is given by translation into simpler constructs. If these translations are applied exhaustively, the result is a program written in a small subset of Haskell that we call the Haskell *kernel*.

Although the kernel is not formally specified, it is essentially a slightly sugared variant of the lambda calculus with a straightforward denotational semantics. The translation of each syntactic structure into the kernel is given as the syntax is introduced. This modular design facilitates reasoning about Haskell programs and provides useful guidelines for implementors of the language.

1.3 Values and Types

An expression evaluates to a *value* and has a static *type*. Values and types are not mixed in Haskell. However, the type system allows user-defined datatypes of various sorts, and permits not only parametric polymorphism (using a traditional Hindley-Milner type structure) but also *ad hoc* polymorphism, or *overloading* (using *type classes*).

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press

Errors in Haskell are semantically equivalent to $\perp$. Technically, they are not distinguishable from nontermination, so the language includes no mechanism for detecting or acting upon errors. However, implementations will probably try to provide useful information about errors (see Section 3.1).

1.4 Namespaces

There are six kinds of names in Haskell: those for *variables* and *constructors* denote values; those for *type variables*, *type constructors*, and *type classes* refer to entities related to the type system; and *module names* refer to modules. There are two constraints on naming:

1. Names for variables and type variables are identifiers beginning with lowercase letters or underscore; the other four kinds of names are identifiers beginning with uppercase letters.
2. An identifier must not be used as the name of a type constructor and a class in the same scope.

These are the only constraints; for example, `Int` may simultaneously be the name of a module, class, and constructor within a single scope.

https://doi.org/10.1017/S0956796803000315 Published online by Cambridge University Press