My Reference Book

Rahul Kadukar,
North Bergen, New Jersey,
USA 07047

rahul.kadukar@rutgers.edu
kadukar.rahul@gmail.com

November 27, 2016

Contents

PREFACE

This book is about implementing functional programming languages using graph reduction.

Functional languages have become the focus of much active research in recent years [Backus, 1978] [Peyton Jones, 1984], but their acceptance has been delayed by the inefficiency of their available implementations when compared with more conventional languages.

This situation has changed recently, with the advent of rather fast implementations of functional languages such as Cardelli's ML [Cardelli, 1983], Fairbairn's Ponder [Fairbairn, 1982], and the Chalmers Lazy ML compiler [Johnsson, 1984]. These implementations rival the speed Of compilers for more conventional languages.

There appear to be two main approaches to the efficient implementation of functional languages. The first is an environment-based scheme, exemplified by Cardelli's ML implementation, which derives from the experience of the Lisp community. The other is graph reduction, a much newer technique first invented by Wadsworth [Wadsworth, 1971], and on which the Ponder and Lazy ML implementations are founded. Despite the radical differences in beginnings, the most sophisticated examples of each approach show remark- able similarities.

The techniques of graph reduction are to be found scattered amongst the proceedings of various conferences and workshops, and it is one purpose of this book to collect some of this work together. book is intended to have two main applications:

(i) As a course text for part of an undergraduate or postgraduate course on the implementation of functional languages.
(ii) As a handbook for those attempting to write a functional language implementation based on graph reduction.

The material is presented in a fairly informal tutorial fashion, the idea being to convey the framework and some of the intuitions that will render the original sources more accessible.

Chapters 5 and 7 were written by Philip Wadler, Of the Programming Research Group, Oxford, and Chapter 4 was written in collaboration with him.

Chapters 8 and 9 were written by Peter Hancock, of Metier Management Systems Ltd, and currently at the Programming Research Group, Oxford. I gratefully acknowledge their patience in writing and rewriting their drafts.

I am extremely grateful to a number Of other people who have made significant technical contributions to the book. David Turner's help was invaluable, in offering comments on the parts of the book that relate to Miranda. The Appendix which gives an introduction to Miranda, was written by him. (Miranda is a trademark of Research Software Limited.)

Much of the information about Miranda in the first part of the book is based on a prerelease version of the Miranda system, and I am grateful to Research Software Limited for the permission to include this material.

Simon Finn, Of the University of Stirling, made a number of penetrating observations about the treatment of pattern-matching, which resulted in significant improvements in Chapters 4 and 6.

Several long discussions with Thomas Johnsson, of Chalmers University, Goteborg, radically changed the shape of the G-machine chapters, and Chapter 21 was written in collaboration with him. John Fairbairn and Stuart Wray, of Cambridge, also made important contributions in this area.

Paul Hudak and Robert Keller helped me in writing the last section of Chapter 24.

Many other people have given me welcome help and encouragement, and have helped enormously by reading the text and making constructive suggestions. These include Lennart Augustsson, Philip Boswell, Geoff Burn, Nigel Chapman, Chris Clack, Pierre-Louis Curien, Dan Friedman, Kevin Hammond. Peter Hancock, Chris Hankin, John Hughes, Thomas Johnsson, Dick Kieburtz, Phil Molyneux, Benedict Nixon, Martin Packer, Ellen Poon, Jon Salkild, William Stoye, David Turner, Phil Wadler, John Washbrook and Russel Winder- I am particularly grateful to John Washbrook for his unfailing support.

Simon L Peyton Jones
University College London
London WCIE 6BT
simonpj@uk.ac.uel.cs

References

Backus, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra Of programs. *Communications Of the ACM.* Vol. 21. no. 8, pp. 613-41.

Cardelli, L. 1983. The functional abstract machine. *Polymorphism. Vol. 1*, no. l.

Fairbairn, J. 1982. Ponder and its type system. *Technical Report 31.* Computer Lab.. Cambridge. November.

Johnsson, T. 1984. Efficient compilation of lazy evaluation, In *proceedings of the ACM Conference on Compiler Construction*, Montreal, pp. 58-69. June.

Peyton Jones, S.L. 1984. Directions in functional programming research. in *SERC Distributed Computing Systems*, pp. 220-49. Duce (editor). Peter Peregrinus.

Wadsworth, Cp. 1971. Semantics and pragmatics of the lambda calculus. Chapter 4. PhD thesis, Oxford.

Part I

One

Chapter 1

Introduction

This book is about implementing functional programming languages using lazy graph reduction, and it divides into three parts.

The first part describes how to translate a high-level functional language into an intermediate language, called the lambda calculus, including detailed coverage of pattern-matching and type-checking. second part begins with a simple implementation of the lambda calculus, based on graph reduction, and then develops a number of refinements and alternatives, such as super- combinators, full laziness and SK combinators. Finally, the third part describes the G-machine, a sophisticated implementation of graph reduction , which provides a dramatic increase in performance over the implementations described earlier.

One of the agreed advantages of functional languages is their semantic simplicity. This simplicity has considerable payoffs in the book. Over and over again we are able to make semi-formal arguments for the correctness of the compilation algorithms, and the whole book has a distinctly mathematical flavor - an unusual feature in a book about implementations.

Most of the material to be presented has appeared in the published literature in some form (though some has not), but mainly in the form of conference proceedings and isolated papers. References to this work appear at the end of each chapter.

1.1 Assumptions

This book is about implementations, not languages, so we shall make no attempt to extol the virtues of functional languages or the functional programming style. Instead we shall assume that the reader is familiar with functional programming; those without this familiarity may find it heavy

going. A brief introduction to functional programming may be found in Darlington [1984], while Henderson [1980] and Glaser *et al.* [1984] give more Substantial treatments. Another useful text is Abelson and Sussman [1985] which describes Scheme, an almost-functional dialect of Lisp.

An encouraging consensus seems to be emerging in the basic features of high-level functional programming languages, exemplified by languages Such as SASL [Turner, 1976], ML [Gordon al., 1979], KRC [Turner, 1982], Hope [Burstall *et al.*, 1980], Ponder [Fairbairn, 1985], LML [Augustsson, 1984], Miranda [Turner, 1985] and Orwell [Wadler, 1985]. However, for the sake of definiteness, we use the language Miranda as a concrete example throughout the book (When used as the name of a programming language, 'Miranda' is a trademark of Research Software Limited.) A brief intro- duction to Miranda may be found in the appendix, but no serious attempt is made to give a tutorial about functional programming in general, or Miranda in particular. For those familiar with functional programming, however, no difficulties should arise.

Generally speaking, all the material of the book should apply to the other functional languages mentioned, with only syntactic changes. The only exception to this is that we concern ourselves almost exclusively with the implementation Of languages with non-strict semantics (such as SASL, KRC, Ponder, LML, Miranda and Orwell). The advantages and disadvantages of this are discussed in Chapter 11, but it seems that graph reduction is probably less attractive than the environment-based approach for the implementation Of languages with strict semantics; hence the focus on non-strict languages. However , some functional languages are strict (ML and Hope, for example) , and while much of the book is still relevant to strict languages, some of the material would need to be interpreted with care.

The emphasis throughout is on an informal approach, aimed at developing understanding rather than at formal rigor. It would be an interesting task to rewrite the book in a formal way, giving watertight proofs of correctness at each Stage.

1.2 Part I: Compiling High-level Functional Languages

It has been widely observed that most functional languages are quite similar to each other, and differ more in their syntax than their semantics. In order to simplify our thinking about implementations, the first part of this book shows how to translate a high-level functional program into an intermediate language which has a very simple syntax and semantics. Then, in the second and third parts of the book, we will show how to implement this intermediate language using graph reduction. Proceeding in this way allows us to describe graph reduction in considerable detail, but in a way that is not specific to any particular high-level language.

The intermediate language into which We will translate the high-level

functional program is the notation of the lambda calculus (Figure 1.1). The lambda calculus is an extremely well-studied language, and we give an introduction to it in Chapter 2.

The lambda calculus is not only simple, it is also sufficiently expressive to allow us to translate any high-level functional language into it. However, translating some high-level language constructs into the lambda notation is less straightforward than it at first appears, and the rest of Part I is concerned with this translation.

Part I is organized as follows. First of all , in Chapter 3, we define a language which is a superset Of the lambda calculus, which we call the enriched lambda calculus. The extra constructs provided by the enriched lambda calculus are specifically designed to allow a straightforward translation of a Miranda program into an expression in the enriched lambda calculus, and Chapter 3 shows how to perform this translation for simple Miranda programs.

After a brief introduction to pattern-matching, Chapter 4 then extends the translation algorithm to cover more complex Miranda programs, and gives a formal semantics for pattern-matching. Subsequently, Chapter 7 rounds out the picture, by showing how Miranda's ZF expressions can also be translated in the same way. (Various advanced features of Miranda are not covered, such as algebraic types with laws, abstract data types, and modules.)

Much of the rest of Part I concerns the transformation of enriched lambda calculus expressions into the ordinary lambda calculus subset, a process which is quite independent of Miranda. This language-independence was one of the reasons for defining the enriched lambda calculus language in the first place.

Chapter 5 shows how expressions involving pattern-matching constructs may be transformed to use case-expressions, with a considerable gain in efficiency. Then Chapter 6 shows how all the constructs of the enriched lambda calculus, including case-expressions, may be transformed into the ordinary lambda calculus.

Part I concludes with Chapter 8 which discusses type-checking in general, and Chapter 9 in which a type-checker is constructed in Miranda.