Functional Programming

and

Parallel Graph Rewriting

Rinus Plasmeijer

Marko van Eekelen

Preface

Functional languages

In a **declarative programming language** a computation is expressed in a static fashion, as a list of declarations. A program in such a language is regarded as a specification that happens to be executable as well. In this textbook we focus on a subclass of the declarative languages, the **functional programming languages**, sometimes called **applicative languages**. In these languages a program consists of a list of function definitions. The execution of a program consists of the evaluation of a function application given the functions that have been defined.

In this book we want to show that functional programming languages can be used to write compact, reliable and readable programs. As example languages we shall use the functional languages Miranda and Concurrent Clean. We shall explain the advantages and disadvantages of functional languages, show how efficient implementations can be obtained on sequential and parallel architectures, and discuss the important underlying theoretical frameworks. The expressive power of functional languages is shown not only by some small examples. The descriptions, transformation and compilation schemes, as well as the definitions of the abstract machine architectures given in this textbook are *all* presented in a functional style illustrating the suitability of functional languages as specification tools.

Graph rewriting systems

The concept of a function is a fundamental notion in mathematics. One of the very strong points of functional languages is the property that their semantics is based very directly on well-defined mathematical models of computation. In fact, functional languages can be regarded as sugared versions of these models. These models are very important because they abstract from unimportant language details. In this way they help to get a better understanding of the essential underlying concepts of a language. This knowledge can then be used to improve the language itself, but it can also be used to make better implementations.

The traditional model of computation used for functional languages is the **λ-calculus**. Field and Harrison (1988) give a very good, thorough survey of several functional languages and their implementations using the λ-calculus as the basic model of computation. They use HOPE as a functional language instead of Miranda and they restrict themselves mainly to sequential implementations.

This book differs from other textbooks in that it focuses on an alternative model of computation: **graph rewriting systems**. Like the λ-calculus this model has the advantage that it is simple and that it can model functions. Unlike the λ-calculus it additionally contains the important notions of patterns and graphs. Graphs (and the related notion of sharing) play a key role in any efficient (parallel) implementation. To reason about the computational complexity of a program it is essential to know whether computations are shared or not. Furthermore, the notion of uniqueness in graphs gives an efficient solution for modelling interaction and side-effects within a pure functional framework.

Implementation on sequential architectures

The implementation of functional languages is quite complex when efficient code has to be generated. In this book we shall explain how such an implementation can be achieved on traditional sequential architectures. The implementation of functional languages is explained in a number of steps. Starting from Miranda, we go down in level of abstraction until we reach the concrete machine level. In this compilation process the functional language Concurrent Clean (based upon graph rewriting systems) is used as intermediate language.

Concurrent programming and parallel evaluation

Another advantage of functional languages is that they are, in principle, suited for concurrent (parallel or interleaved) evaluation. However, an implementation on parallel hardware is much more difficult than on sequential hardware, mainly because parallel machines are hard to program anyway. Concurrent functional programming is being investigated at several research institutes but is still in its infancy. We shall demonstrate its potential power and discuss the implementation on loosely coupled parallel architectures.

About this book

This book is intended as a textbook for a one or two semester (term) course: an *introductory course* on functional programming, models of computation and the implementation of functional programming languages on traditional sequential machine architectures (Parts 1, 2 and 4), optionally followed by an *advanced course* focused on analysis methods (strictness analysis and type theory), concurrent functional programming and implementation on parallel architectures (Parts 3 and 5).

The prerequisite for this book is an introductory course in computer science including knowledge of programming in a high-level language as well as some basic understanding about (parallel) computer architectures and compiler construction.

This textbook is suited for advanced undergraduate students, for postgraduate students of computer science and for researchers interested in the area of functional programming and the implementation techniques on sequential and parallel architectures.

The subject of this textbook is a hot and very interesting research topic. It is not an easy topic because many aspects of computer science play a role in it. Although much progress has been made during the last decade, also many problems are still to be solved and are expected to remain unsolved for quite a while. Instead of covering all existing and proposed lines of research we have chosen to base this book mainly on the work performed by our research team at the University of Nijmegen. This has the advantage that one complete line of approach, involving theory, specification and implementation, can be described in detail based upon actual experience. For example, all translation schemes presented in this book are abstracts from the prototype specifications written in a functional formalism that have been used as blueprint for the actual implementations. The approaches explained in this book have produced good results. Furthermore, they have many things in common with other approaches, so one will still get a very good overall insight on the actual problems and how they can be handled.

About the Concurrent Clean software

The Concurrent Clean system, the software system developed at the University of Nijmegen using the approaches described in this textbook, can be ordered separately via the authors (from the address at the end of the Preface). This software can be used to obtain practical experience in (concurrent) functional programming. With this system one can develop real applications.

The Concurrent Clean system contains a programming environment (including text editor and project manager), an interpreter and a code generator for the lazy functional language Concurrent Clean (Brus *et al.,* 1987; Nöcker *et al.,* 1991b). The system is especially designed for the Apple Macintosh (Plasmeijer *et al.,* 1991) and generates efficient code for any Mac. An I/O-library is available offering a very high-level specification of modern I/O-handling (windows, dialogues and the like). Furthermore, it is possible to generate parallel code for Macs connected in a local area network (AppleTalk/Ethernet).

Concurrent Clean and the Concurrent Clean system are also available on other platforms. The Macintosh version runs on an Atari ST (using a Mac emulator). There is a version for Sun3 (Motorola) and Sun4 (SPARC) under UNIX using an X Window System with Open Look or OSF-Motif for the low-level I/O-handling. Clean programs can run on any of these platforms without the need to modify source code. A cross-compiler is available for ParSyTec's Supercluster (transputer). A version for the IBM-PC (look-alikes) is under development.

The Clean research team is working on the improvement of Concurrent Clean and the Concurrent Clean system. Please contact the authors for the latest information.

Outline of this book

This book is divided into five parts. In the first two parts we focus on *functional programming* and suitable underlying *models of computation*. Then, in Part 3 these models are used to show how properties of functional programs can be *analysed*. Part 4 uses a particular model to show in detail how functional programs can be *implemented* efficiently on traditional architectures. The final part (Part 5) covers the same kind of subjects as Parts 1, 2 and 4 respectively, but focuses on parallelism. Since the research in this area is less stabilized, this final part is of a more introductory and reflective nature. Exercises are given at the end of each chapter (difficult or large exercises are marked *).

Part 1: Functional programming

In this first chapter we give a description of the *basic language concepts* that most functional languages have in common (Chapter 1). Hereafter we show in more detail the descriptive power and syntactic sugar of the *advanced language concepts* as offered by one particular functional programming language: Miranda (Chapter 2).

Part 2: Models of computation

In this part we concentrate on three models of computation: *λ-calculus* (Chapter 3) as the best known computational model and *term rewriting systems* (Chapter 4) and *graph rewriting systems* (Chapter 5) as alternative models of computation. The latter is used throughout this book as a basis for implementations and for reasoning about efficiency aspects.

Part 3: Analysis of functional programs

The models of computation introduced in Part 2 are used in Part 3 to show how additional information can be obtained by static analysis. In particular, type assignment systems (Chapter 6) and strictness analysis (Chapter 7) are discussed. Type and strictness information is needed to obtain efficient implementations. This theoretical part might be skipped by readers with a greater interest in more practical subjects.

Part 4: Implementation on sequential architectures

This part of the book covers the implementation of functional languages on traditional machine architectures. The implementation is discussed in a number of steps. In each step we go down in level of abstraction until the concrete machine level is reached (Figure 0.1).

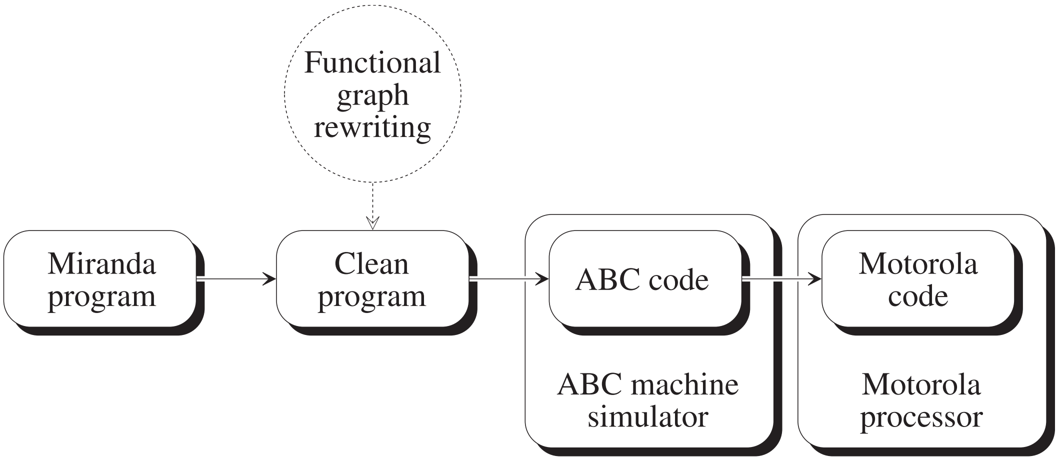

Figure 0.1 Sequential implementation of functional programming languages in four levels of abstraction.

The language *Clean* is introduced in Chapter 8. This language is based on *functional* graph rewriting systems. It is shown how a Miranda program can be translated into an equivalent Clean program (Chapter 9). Hereafter an abstract machine, the *ABC machine*, is introduced modelling traditional stack-based machine architectures (Chapter 10). The use of an abstract machine prevents us from getting lost in the details of concrete architectures. Step by step we shall deduce compilation schemes from Clean to the instruction set of the ABC machine (Chapter 11). Finally, the remaining problems of an actual implementation on concrete target machines are discussed (Chapter 12). As a representative target architecture the Motorola processor is used.

Part 5: Concurrency issues

There are several ways in which concurrency can be exploited in a functional program. Because loosely coupled parallel architectures are becoming widely available we concentrate on the implementation on this type of architecture. To get an optimal performance on these machines the parallelism should be *explicitly* specified by the programmer.

None of the existing well-known functional languages have language facilities to control parallelism. Therefore we introduce a possible extension to express concurrency in functional languages (Chapter 13). How the underlying model of computation can be extended to capture the notion of processes and process communication is the subject of Chapter 14.

Finally, the implementation of functional languages on loosely coupled parallel machine architectures is discussed. As in Part 4 the implementation is explained through a series of abstraction steps down to the concrete machine level (Figure 0.2).

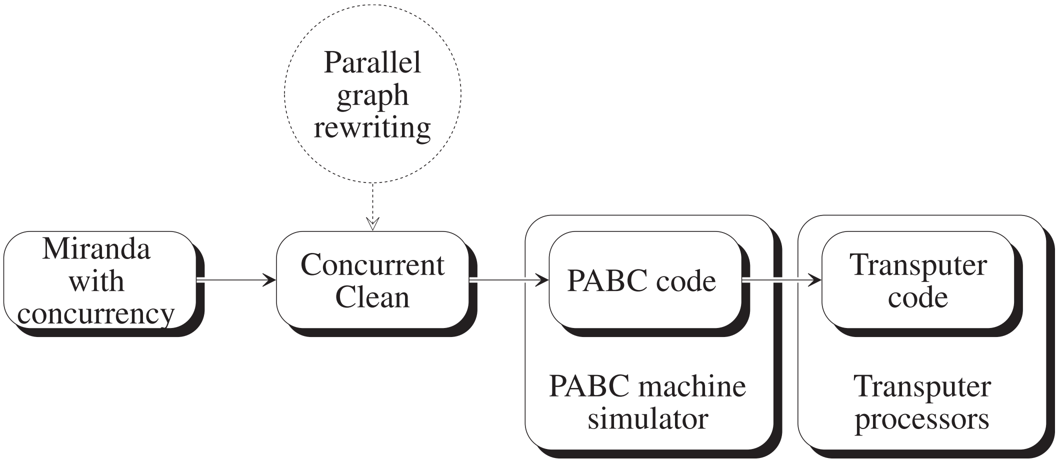

Figure 0.2 Parallel implementation of functional programming languages in four levels of abstraction.

First, we introduce the extension of *Clean* to *Concurrent Clean* and show how a concurrent functional program can be translated into an equivalent *Concurrent Clean* program (Chapter 15). Then, we extend the ABC machine (Chapter 10) to a *parallel ABC machine* modelling loosely coupled machine architectures and show how Clean's concurrency primitives are translated to abstract machine instructions (Chapter 16). Finally, we discuss the remaining problems of an actual implementation (Chapter 17) on a concrete parallel target machine. As a representative target architecture the transputer processor is used.

Acknowledgements

The following members of the Nijmegen research team have made important contributions to specific aspects of this book: Peter Achten (Clean's event I/O), Steffen van Bakel (type assignment systems), John van Groningen (code generators and machine wizarding), Halbe Huitema (I/O library, testing), Marco Kesseler (transputer code generator), Pieter Koopman (transformation schemes, (P)ABC machine), Eric Nöcker (transformation schemes, strictness analysis, PABC machine), Sjaak Smetsers (transformation schemes, Concurrent Clean, PFGRSs, (P)ABC machine) and Ronny Wichers Schreur (Concurrent Clean environment).

We would like to thank the following people who each in their own way made an important contribution to the realization of this book and to the design and the construction of the Concurrent Clean system: Christ Aarts, Erik Barendsen, Henk Barendregt, Tom Brus, Cor de Cloe, Guido van Dam, Wil Dekkers, Maurice Klein Gebbink, John Glauert, Johan Goos, Harco de Hilster, Martin van Hintum, Jeroen ter Hofstede, Richard Kennaway, Mathieu Knops, Hans Koetsier, Frank van Latum, Maarten van Leer, Mieke Massink, Paul de Mast, Angelo Melis, Betsy Pepels, Leon Pillich, Ab Reitsma, Richard van Schelven, Ronan Sleep, Daniël Tuynman, Marten Wensink, Koen Wouters and Mariëlle van der Zandt.

We appreciate very much the work of the referees who have helped us with several very good suggestions for improvement of this textbook.

We are very grateful for the financial injection of the *'Dienst Wetenschapbeleid'* of the Dutch *Department of Science and Education* which initiated the research that forms the basis for this textbook. Furthermore, we would like to thank the *ESPRIT* projects *Tropics* and *Semagraph* for supporting the more advanced parts of that research.

Rinus Plasmeijer and Marko van Eekelen

April 1993

Computer Science Department
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

email: rinus@cs.kun.nl
marko@cs.kun.nl

To Trees and Lianne

Contents