Chapter 1
Introduction

This chapter contains both an overview of the problem area to be discussed and an overview of the structure of the rest of the book.

1.1. OVERVIEW OF THE PROBLEM AREA

Functional programming provides a new way of writing programs and a new way of thinking about problem solving (Backus 1978). A specific advantage of functional programs is the fact that they are easy to reason about, since they can be viewed as mathematical specifications of algorithms, and are therefore amenable to automatic verification techniques. Also, there is a belief in some circles that functional programs are easier to write than other programs. This is because functional programming languages provide powerful higher-order composition mechanisms which are not found in conventional imperative languages such as C. Furthermore, the combination of these mentioned qualities can lead to reliable software systems (Hughes 1984). Although the foundations of functional programming have been known for some time (Curry & Feys 1968, Landin 1966, Reynolds 1972), most of what we know about the field has been discovered in the last ten years. Therefore, the potential benefits of using functional programming techniques are still largely unexplored.

Lazy evaluation (Henderson & Morris 1976, Freedman & Wise 1976) of functional programs allows the use of powerful programming structures such as implicit coroutining and infinitely long lists. Unfortunately, the power and flexibility of lazy evaluation has, in the past, been associated with extreme inefficiency when executing programs. It is common for programs to be 100 times slower in a lazy functional language than in an imperative language such as C.* Because programs written in these languages execute so slowly, it is difficult to build a large software base to gain experience in using the languages. And,

* Actual comparisons will be given in a later chapter.

without a large software and user base, it will be difficult to gain insights on the appropriateness of lazy functional programming languages for solving real problems.

One important evaluation strategy for lazy functional programming languages is *graph reduction*. Graph reduction involves converting the program to a lambda calculus expression (Barendregt 1981), and then to a graph data structure. One method for implementing the graph data structure is to translate the program to combinators (Curry & Feys 1968). A key feature of this method is that all variables are abstracted from the program. The program is represented as a computation graph, with instances of variables replaced by pointers to subgraphs which compute values. Graphs are evaluated by repeatedly applying graph transformations until the graph is irreducible. The irreducible final graph is the result of the computation. In this scheme, the rewriting of the graph data structure, also called *combinator graph reduction*, is the method used to execute the program.

A great allure of combinator graph reduction is that it may provide an automatic approach to parallel computation, since the available parallelism of a program compiled to a graph is directly represented by the graph structure (Peyton Jones 1987). Such parallelism tends to be fine-grained, where each quantum of work available is small in size. Overhead in managing resources and task scheduling can quickly dominate the performance of a fine-grained parallelism system, so it is important to find a scheme in which overhead is kept low to achieve reasonable speedups.

Traditionally, it has been assumed that advanced programming languages (and in particular functional programming languages) require radically different, non-vonNeumann architectures for efficient execution. This book explores mapping functional programming languages onto conventional architectures using a combination of techniques from the fields of computer architecture and implementation of advanced programming languages.

The tools of the computer architect shed new light on the behavior of this special class of programs. *The results shown here suggest that the advanced programming languages being explored by computer scientists do not adhere to the normal expectations of computer architects*, and may eventually force a reevaluation of architectural tradeoffs in system design. An important point of the findings presented here is that the combination of architectural features required for efficiency may be relatively inexpensive, yet omitted from even recent machines because of relative unimportance for conventional programming language execution.

1.2. ORGANIZATION OF THIS BOOK

The book examines existing methods of evaluating lazy functional programs using combinator reduction techniques, implementation and characterization of a means for accomplishing graph reduction on uniprocessors, and analysis of the potential for special-purpose hardware implementations.

Chapter 2 provides a background on functional programming languages and existing implementation technology. The reader who is not familiar with the field may wish to read Appendix A, which is a tutorial on combinator graph reduction. Chapter 2 also contains a summary of important previous work on the combinator reduction approach to evaluating lazy functional programming languages.

Chapter 3 describes the TIGRE methodology for implementing combinator graph reduction. The description is in the form of a progression of techniques which are added to a graph reduction mechanism based on previously used methods. The general flow of the incremental improvements starts with conventional graph reduction methods, moves on to a fast interpretation scheme for combinator graphs, refines the method to a direct execution scheme for combinator graphs, and then discusses supercombinator compilation methods for improved performance.

Chapter 4 describes the TIGRE abstract machine, which is used to implement the graph reduction methodology described in Chapter 3. TIGRE may be described in terms of an abstract architecture and abstract assembly languages. These abstract definitions have been mapped efficiently onto real languages and architectures, including machine-independent C code and assembly language implementations for the VAX family and the MIPS R2000 processor.

Chapter 5 gives the results of performance measurements of TIGRE on a variety of platforms. These results are compared with available results for other combinator reduction strategies and against the performance of imperative languages.

Chapter 6 discusses architectural metrics for TIGRE executing on the MIPS R2000 processor. The architectural metrics include a simulation of cache behavior, combinator execution frequency, and various dynamic metrics such as heap allocation statistics.

Chapter 7 explores the potential for special-purpose hardware to yield further speed improvements. In order to maintain some basis in reality, modifications to the MIPS R2000 architecture as implemented in the DECstation 3100 platform are proposed, along with predicted speed improvements.

Chapter 8 summarizes the results of the research, and suggests areas for further investigation. While the TIGRE method of graph reduction offers substantial performance improvements over several other existing methods, more work in the areas of compiler technology and parallel implementation is needed.