1

Introduction

This text looks at computer organization and architecture from a strictly conceptual point of view. It is primarily concerned with ways and means of organizing computations, emphasizing the relationship between algorithmic problem specifications and the very basic mechanisms and runtime structures necessary to transform these specifications step by step into problem solutions. We will completely abstract both from concrete programming languages, whether imperative or functional, and from concrete machine architectures, their instruction sets, data formats, addressing modes, register sets, etc., and nothing will be said about their hardware implementation either. Only in the last chapter will a brief overview of two representative real machine architectures be given to show how they relate to the various abstract machines we are going to talk about.

These abstract machines form what may be considered common interfaces that may be shared by real computing machines featuring widely varying architectures. They are derived from basic theoretical concepts of computer science that are invariant against actual trends of doing things. These concepts were originally developed in response to the fundamental question of what can be effectively computed in principle and of what the basic mechanisms for having these computations performed by machinery are.

Computability became a subject of intensive research between 1930 and 1940, interestingly enough, some time before the first computers as we know them today came into being. It led to various mathematical models, developed more or less independently, that capture in a nutshell the essence of performing computations mechanically. These models include Post's production systems, Markov algorithms, Kleene's recursive functions, Schoenfinkel's and Curry's combinators, Church's λ-calculus and the Turing machine, all of which are equivalent with respect to provable propositions about what can and what cannot be accomplished with algorithmic approaches to problem solving. Though more than 60 years old by now, these models are lasting and stable foundations of computer science.

The preferred model for studying computability is the Turing machine, since, on a very elementary level, it closely mimics the workings of computers. The machine consists of a tape that holds sequences of characters from some finite alphabet (including blanks) and a primitive processor that can be moved back and forth along the tape. It also includes controls that may assume one of finitely many states. The machine goes repeatedly through a cycle of transforming current states and characters read from the tape into next states and characters written on the tape, and of moving the processor by one character position to the left or right. Disregarding efficiency, this primitive apparatus is capable of computing solutions for all problems that can be specified algorithmically. It is a simple model of what is doable in principle by any existing or yet to be invented computing machine.

However, the controls that need to be 'wired' into the processor to do the jobs at hand bear hardly any resemblance to algorithms as they may be specified in some high-level language. Programming the Turing machine is primarily concerned with organizing computations as sequences of elementary character manipulations, using the very basic mechanisms of substituting one thing (a character) by another one and of moving along the tape to the positions where substitutions have to take place. Though these two mechanisms realize the most important operations of computing, more important than adding numbers, the level of granularity is simply too fine to relate the workings of the Turing machine in an easily comprehensible way to the computational steps specified by high-level algorithms.

The computational model that bridges the gap between high-level algorithmic specifications and the machines that are capable of executing them is the λ-calculus. It strongly influences today's programming paradigms, the basic operating principles of computing machines, the runtime environments that need to be built up during program execution, and to some extent also the design of compilers that translate high-level algorithms into machine codes.

The λ-calculus is a theory of computable functions. It talks about elementary properties of operators and operands, about the application of operators to operands and about the role of variables in this game. In its simplest and purest form, the λ-calculus knows only three syntactical figures for the construction of computable expressions – variables, abstractions (of variables from expressions) and applications (of operator to operand expressions) – and a single rule for transforming λ-expressions into other λ-expressions. This β-reduction rule, which specifies the substitution of variables by λ-expressions, tells the whole story about computing, and it does so in a more appropriate setting than the simple character substitutions of the Turing machine.

In this text, we will therefore take the λ-calculus as the starting point for a tour through various abstract computing machines. This tour leads from complete realizations of the λ-calculus to restricted forms of it that can typically be found in implementations of functional and imperative languages. The idea is to follow what is commonly known as a language-directed approach toward architecting computing machines that emphasizes the basic mecha-

nisms and runtime structures necessary to perform algorithmically specified computations, rather than the handling of bits, bytes, addresses, etc. on the register-transfer structure of concrete hardware machinery.

In this text we will proceed as follows.

Chapter 2 discusses rather informally some essentials of designing and executing algorithms. They include the concepts of variables and of (recursive) abstractions, the termination problem, symbolic computations, and operations on structured data. The chapter also gives an overview of types and type systems. The purpose of the chapter is to highlight some issues that require a more formal treatment in subsequent chapters, since they play an important role in designing abstract machines.

Chapter 3 introduces an expression-oriented algorithmic language AL that will be used as a reference language throughout the text. Its semantics is defined by an abstract evaluator that prescribes how and in what order the value of an expression may be computed from the values of its subexpressions. The chapter identifies some problems related to the chosen evaluation strategy and also to the freedom provided by the AL syntax for designing algorithms.

To fully understand the implications of these problems requires a close look at the underlying theory, which is given in Chap. 4 on the λ-calculus. It begins with a precise definition of the binding status of variables and of the β-reduction rule that governs the substitution of variables by expressions, including the orderly resolution of potential naming conflicts. The unbinding mechanism used to this effect leads to a nameless Λ-calculus that represents binding structures by means of indices that considerably facilitate the implementation of the β-reduction rule.

The chapter also discusses reduction strategies such as applicative (operands-first) versus normal (operands-when-needed) order, confluence, termination with full normal forms (which are the ultimate goals of reducing λ-expressions), and with intermediate head normal forms and weak (head) normal forms. It also addresses recursions in the λ-calculus, outlines how the pure λ-calculus may be extended by primitive arithmetic, logic and relational operations on numbers, Boolean values and character strings and by operations on simple structured data, and formalizes what has been said about typing in Chap. 2.

This brief excursion into the λ-calculus covers everything that needs to be known to understand the abstract machines described in the following chapters.

In Chap. 5, we begin with a very simple abstract machine that interprets expressions of the pure λ-calculus. This machine, which supports both applicative- and normal-order reduction, is derived from Landin's original SECD machine. It is a weakly normalizing machine, meaning that it implements a naive form of β-reduction that does not penetrate abstractions. An important concept realized by this machine is that of delayed substitutions using an environment. Closely related to this concept are closures that pair abstractions with the environments in which they may have to be evaluated

later on. Delayed substitutions, environments and closures are the key ingredients of efficient computations that are shared by several of the abstract machines discussed in subsequent chapters.

The chapter also includes brief descriptions of two other weakly normalizing abstract machines, the $\mathcal{K}$-machine and the categorial abstract machine.

Chapter 6 describes two interesting approaches to fully normalizing machines that employ environment-based β-reductions.

The $\lambda\sigma$-calculus introduces environments through the notion of explicit substitutions, as an extension of the nameless Λ-calculus. These substitutions are manipulated by a set of σ-rules that in fact define a weakly normalizing abstract machine. Continuing beyond weak normal forms requires a special *beta*-rule that pushes substitutions under abstractions, whereupon weak normalization may be resumed in abstraction bodies. Repeated weak normalizations followed by applications of this *beta*-rule lead to head normal forms, and applying this head normalization to all subexpressions of head normal forms produces full normal forms.

The other approach treats environments as an integral part of the Λ-calculus itself. It is based on the systematic transformation of Λ-expressions from head forms to head normal forms by so-called β-reductions_in_the_large, governed by a head-order reduction regime. This is basically a process that recursively distributes largest possible chunks of consecutive β-redices over the components of head forms, thus in fact creating environments for binding indices that occur in head positions. These indices either select from the environments other expressions with which the process continues in their place or, if they reach beyond the environments, terminate with head normal forms.

Fully normalizing λ-calculus machines that realize both concepts are described in the following four chapters.

Chapter 7 takes the abstract head-order reducer one step closer toward a real machine. It uses graph reduction techniques based on the substitution and rearrangement of pointers, rather than of the (sub)expressions or the environments they represent, which permits a great deal of sharing of the evaluation of subexpressions among several pointer occurrences. It is the key to achieving significantly better runtime efficiency as compared with direct interpretation.

In Chap. 8, this graph reducer is turned into a code-executing abstract machine. As an interesting feature that follows from head-order reduction, this machine supports two instruction streams, of which one executes in a forward direction to dynamically generate the other stream which executes in a backward direction. Both codes in cooperation produce the code-equivalent of fully normalized expressions eventually, if they exist.

The *G*-machine introduced in Chap. 9 is another code-executing abstract machine, specifically designed for the implementation of functional languages with lazy semantics. It is weakly normalizing, permitting the computation of ground terms (or basic values) only, which is more or less a consequence of

compiling functions to static code. This goes hand in hand with the conversion, prior to compilation, of nested function definitions into flat sets of closed abstractions, also referred to as supercombinators, that rule out reductions under abstractions. Supercombinator compilation yields fairly efficient codes whose runtime environments can be accommodated in single, coherent stack frames.

The idea put forth by the $\lambda\sigma$-calculus leads to another abstract machine concept, presented in Chap. 10. It employs compiled graph reduction similar to that of the G-machine for weak normalization and turns control over to a special η-extension mechanism that prepares weak normal forms for further code-controlled reductions under abstractions. The cycle of code execution and η-extensions is repeated until the expressions are fully normalized. This π–RED machinery comes in two variants, of which one realizes a lazy (operands-when-needed) and the other a strict (operands-first) semantics. Again, nested function definitions are closed prior to compilation to code, but this is done in a less rigorous way than in the G-machine to avoid some of the redundancies of supercombinator reductions.

The complete machinery is made to appear to the user as a system that performs high-level transformations of λ-expressions governed by full β-reductions. These transformations may, under interactive control, be carried out step by step, and intermediate expressions may be displayed to the user in high-level notation for inspection or modification.

Chapter 11 introduces the concept of pattern matching – an operation that extracts (sub)structures from given structural contexts and substitutes them for placeholders in other (structural) contexts. Pattern matching may be effectively employed to quickly prototype, on a meta-language level, compilers and language interpreters (abstract machines), or to implement term rewrite systems and essential parts of theorem provers. The chapter also describes how pattern matching can be implemented on the lazy variant of the machinery described in Chap. 10.

Chapter 12 returns to a weakly normalizing, code-executing functional machine that is more or less a direct descendant of the original SECD machine. In contrast to the G-machine, it implements an applicative-order (or operands-first) regime and also abandons the concept of supercombinator reduction. Instead, it works with open abstractions, closures and runtime structures similar to those used in the machines of Chaps. 7 and 8. This machine is a perfect target for the compilation of AL, and also for such functional languages as Standard ML and SCHEME that feature an applicative-order semantics.

There is only a relatively small step, though one with considerable consequences, from this SECD_I machine to the code-executing abstract machines for imperative languages that are described in Chap. 13. The essence of executing imperative programs is to effect sequences of incremental changes (updates) on selected entries of the runtime environment. This concept is reflected in assignments to variables that represent values held in the runtime environment but are not values themselves, and in abstractions called procedures

that change their calling environments. Since the semantics of imperative languages demands that procedures be applied to full sets of arguments, there is no need to support closures. As a consequence, the runtime environment can be operated as a stack of activation records for procedure calls. Languages that support nested procedure definitions, such as PASCAL, need to have the activation records linked up in compliance with these nestings. Languages that support only flat procedure definitions, such as C, have the complete environments accommodated in coherent activation records that are stacked up in the order in which they are called but otherwise are completely unrelated to each other, i.e., there are no links, which simplifies implementation and enhances runtime efficiency.

The last chapter gives an overview of two representative architectures of real computing machines. Conceptually, they look very much the same as the abstract machines of Chap. 13. The differences that matter from a machine language (assembler) programmer's point of view relate basically to the resources visible at this level that must be accounted for in ways that go beyond what can be expressed by abstract machine code. The finiteness of physical resources (specifically of register sets), certain bandwidth limitations and to some extent also the mechanics of instruction execution call for a well-balanced compromise between what is conceptually needed to support procedure calls and the instruction sets, data formats and memory-addressing modes that should (or can) actually be implemented.

One of the machines described in the chapter features a CISC (complex instruction set) architecture very similar to that of the MC680x0 family. Its instruction set and, specifically, its addressing modes are fairly high level, tailored to the needs of languages that support open procedures that may be nested inside each other, with variable occurrences bound nonlocally in surrounding contexts. It calls for memory-resident runtime environments (stacks) that have their activation records statically linked according to nesting levels.

The other machine belongs to the SPARC family, which has a RISC (reduced instruction set) architecture. Its most interesting feature is a register file that is partitioned into several windows that accommodate the activation records of procedure calls. The windows partially overlap, so that the registers used by a calling procedure to pass parameters are shared with the called procedure. The important point is that only the window of the procedure call that is active is visible at any time; all other windows are inaccessible, and there can be no links to them either, meaning that all the variable instantiations of a procedure call must be packed into a single window.

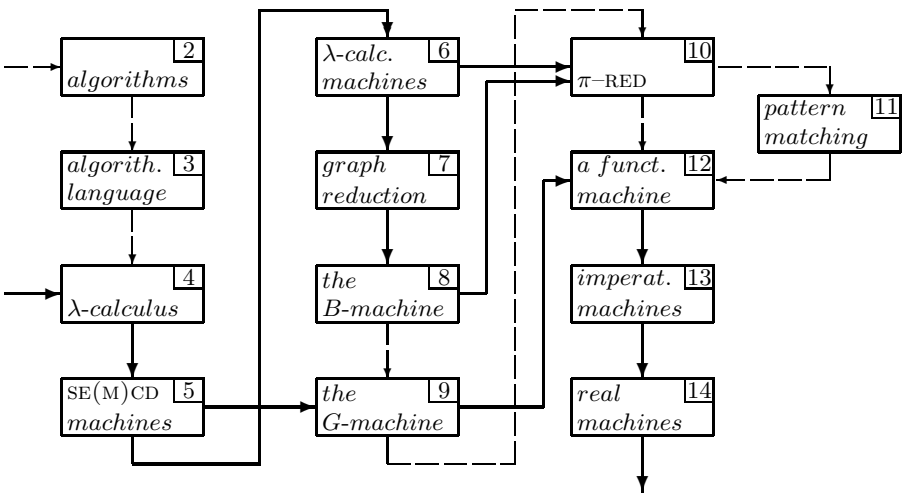
This approach is clearly derived from the programming language C, which knows only flat procedure definitions. Except for references to global variables, these definitions are closed and thus are perfect candidates for direct compilation to code that makes efficient use of the windows.

The text is augmented by two appendices whose contents are somewhat peripheral to the main topic. The first one deals with input/output in expression-oriented languages. It briefly discusses such concepts as interactions with an

external state via streams or environment passing, continuations, and monad-style specifications of interactions. The second appendix addresses theorem proving. It describes, largely by means of a simple example, the basics of a proof process and its AL implementation, which uses pattern matching as introduced in Chap. 11 and normalization of λ-terms to implement the proof rules.

The material included in this text has been used in various ways by the author to teach graduate courses on abstract computing machines. The objective of these courses was to familiarize students with the basic principles of organizing mechanized computations as they are derived from theory. This was thought to be more relevant with regard to understanding the architectures and the workings of computers than talking about the manipulation of bits and bytes in actual hardware machinery. The text is structured roughly as presented in class, though not every subject could be addressed in as much detail. The contents of some of the chapters more or less build on top of other chapters. At the end of each chapter is a summary of its contents.

The diagram below depicts some alternative sequences in which the chapters may be read. Following the thick arrows should give a coherent picture of either weakly or fully normalizing machines. The dashed arrows depict links between descriptions of either kind of machine and also connect to chapters that may be skipped.[1]



The hard core of the text is contained in Chap. 4 on the λ-calculus, specifically in Sect. 4.4 on the nameless Λ-calculus, which is heavily used later on,

[1] The appendices are not included in this diagram.

and in Chaps. 5 and 6 on the basics of weakly and fully normalizing λ-calculus machines. The (graph) reduction machines of Chaps. 7 and 8 can hardly be understood without having read Sect. 6.4 on head-order reductions; the π–RED machines of Chap. 10 follow the basic idea of the $\lambda\sigma$-calculus outlined in Sect. 6.3; and the machines of Chaps. 9, 12, 13 and also those of Chap. 14, which are weakly normalizing, are descendents of and inherit essential features from the SE(M)CD machines of Chap. 5.

The introductory Chaps. 2 and 3 may be left aside by readers who are familiar with algorithms and their evaluation. The chapter on pattern matching may be skipped unless one wishes to read the appendix on theorem proving. The appendix on input/output assumes knowledge of the λ-calculus only; it may therefore be read anywhere after Chap. 4.

Owing to the abstract nature of the subject, the text contains many formal specifications relating to the workings of the various machines and to compilation of high-level algorithms to abstract machine code. These sections are marked $*$ or $**$ in their headings to indicate that they are moderately or very difficult to read. However, whenever deemed necessary, the formal apparatus, mainly sets of state transition rules or sets of compilation rules, is also explained verbally to facilitate understanding.

The text does not explicitly include any exercises, but offers an ample number of challenging problems for homework assignments or for a complementary lab course in which some of the abstract machines and compilers may be rapidly prototyped. This can be conveniently done using the pattern-matching facilities of functional or function-based languages such as HASKELL, CLEAN, Standard ML, or KIR – a language developed by the author's group that has been extensively used for this purpose. Compilers or interpreters for these languages are readily available on the Internet and may be downloaded free of charge from

- www.haskell.org/ghc/download.html (for HASKELL),
- www.cs.kun.nl/∼clean/Download/main/main/htm (for CLEAN),
- www.smlnj.org/software.html (for Standard ML),
- www.informatik.uni-kiel.de/∼base (for KIR).

Prototyping could begin with the fairly simple machines of Chap. 5, the abstract $\lambda\sigma$-machine or the HOR machine specified in Chap. 6. Most suitable for a small termproject would be prototyping the more difficult G-machine of Chap. 9, the strict version of the code-executing π–RED machines of Chap. 10 (stripped of the η-extension part), and the SECD_I machine of Chap. 12.

There is also a lot left to do for paper-and-pencil homework assignments. Besides some exercises in reducing λ-terms, particularly of the nameless Λ-calculus to get acquainted with the manipulation of binding indices, there are several opportunities to do formal specifications that have been omitted from the text, e.g., the state transformation rules for the head-order graph reducer of Chap. 7 and the instruction sets of the π–RED machines. On a simpler level, the specification of instruction sets could be completed, where missing,

by instructions that implement primitive arithmetic, logic and relational operations, including operations on lists. Compiling small example programs by hand, using the compilers specified in Chaps. 9, 10, 12 or 13, could provide other worthwhile exercises.

http://www.springer.com/978-3-540-21146-4

Abstract Computing Machines
A Lambda Calculus Perspective
Kluge, W.
2005, XIV, 384 p. 89 illus., Hardcover
ISBN: 978-3-540-21146-4