Texts in Theoretical Computer Science
An EATCS Series

Editors: W. Brauer G. Rozenberg A. Salomaa
On behalf of the European Association
for Theoretical Computer Science (EATCS)

W. Kluge

Abstract Computing Machines

A Lambda Calculus Perspective

With 89 Figures

Springer

Authors

Prof. Dr. Werner Kluge
Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Olshausenstrasse 40b, 24098 Kiel, Germany
wk@informatik.uni-kiel.de

Library of Congress Control Number: 2004117887

ACM Computing Classification (1998): D.3.2, D.3.4, F.3

ISBN 3-540-21146-2 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media
springeronline.com

Cover design: KünkelLopka, Heidelberg
Typesetting: Camera ready by authors
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Printed on acid-free paper SPIN: 10991046 45/3142/YL - 5 4 3 2 1 0

Klaus Berkling

1931 – 1997

Preface

This monograph looks at computer organization from a strictly conceptual point of view to identify the very basic mechanisms and runtime structures necessary to perform algorithmically specified computations. It completely abstracts from concrete programming languages and machine architectures, taking the λ-calculus – a theory of computable functions – as the basic programming and program execution model. In its simplest form, the λ-calculus talks about expressions that are constructed from just three syntactical figures – variables, functions (in this context called abstractions) and applications (of operator to operand expressions) – and about a single transformation rule that governs the substition of variable occurrences in expressions by other expressions. This β-reduction rule contains in a nutshell the whole story about computing, specifically about the role of variables and variable scoping in this game.

Different implementations of the β-reduction rule in conjunction with strategies that define the sequencing of β-reductions in complex expressions give rise to a variety of abstract λ-calculus machines that are studied in this text. These machines share, in one way or another, the components of Landin's SECD machine – a program text to be executed, a runtime environment that holds delayed substitutions, a value stack, and a dump stack for return continuations – but differ with respect to the internal representation of λ-expressions, specifically abstractions, the structure of the runtime environments and the mechanisms of program execution.

This text covers more than just implementations of functional or function-based languages such as MIRANDA, HASKELL, CLEAN, ML or SCHEME which realize what is called a weakly normalizing λ-calculus that uses a naive version of the β-reduction rule. The emphasis is instead on λ-calculus machines that are fully normalizing, using a complete and correct implementation of the β-reduction rule, which includes the orderly resolution of naming conflicts that may occur when free variables are substituted under abstractions. This feature is an essential prerequisite for correct symbolic computations that treat both functions and variables truly as first-class objects. It may, for instance, be

used to advantage in theorem provers to establish equality between two terms that contain variables, or to symbolically simplify expressions in the process of high-level program optimizations.

In weakly normalizing machines, the flavors of a full-fledged β-reduction are traded in for naive substitutions that are simpler to implement and require less complex runtime structures, resulting in improved runtime efficiency. Naming conflicts are consequently avoided by outlawing substitutions under abstractions, with the consequence that only ground terms (or basic values) can be computed. Weakly normalizing machines are therefore the standard vehicles for the implementation of functional or function-based languages whose semantics conform to this restriction. However, they are also used as integral parts of fully normalizing machines to perform the majority of those β-reductions that in fact can be carried out naively. Whenever substitutions need to be pushed under abstractions, a special mechanism equivalent to full β-reductions takes over to perform renaming operations that resolve potential name clashes.

Abstract machines for classical imperative languages are shown to be descendants of weakly normalizing machines that allow side-effecting operations, specified as assignments to bound variables, on the runtime emvironment. These side effects destroy important invariance properties of the λ-calculus that guarantee the determinacy of results irrespective of execution orders, leaving just the static scoping rules for bound variables intact. In this degenerate form of the λ-calculus, programs are primarily executed for their effects on the environment, as opposed to computing the values of the expressions of a weakly or fully normalizing λ-calculus.

This monograph, though not exactly mainstream, may be used in a graduate course on computer organization/architecture that focuses on the essentials of performing computations mechanically. It includes an introduction to the λ-calculus, specifically a nameless version suitable for machine implementation, and then continues to describe various fully and weakly normalizing λ-calculus machines at different levels of abstractions (direct interpretation, graph interpretation, execution of compiled code), followed by two kinds of abstract machines for imperative languages. The workings of these machines are specified by sets of state transition rules. The book also specifies, for code-executing abstract machines, compilation schemes that transform an applied λ-calculus taken as a reference source language to abstract machine code. Whenever deemed helpful, the execution of small example programs is also illustrated in a step-by-step fashion by sequences of machine state transitions.

I have used most of the material of this monograph in several graduate courses on computer organization which I taught over the years at the University of Kiel. Some of the material (Chaps. 2, 3 and the easier parts of Chaps. 4, 5) I even used in an undergraduate course on programming. The general impression was that at least the brighter students, after some time of getting used to the approach and to the notation, caught on pretty well to the message that I wanted to get across: understanding basic concepts and

principles of performing computations by machinery (with substitution as the most important operation) that are invariant against trendy ways of doing things in real computing machines, and how they relate to basic programming paradigms.

Acknowledgments

There are several people who contributed to this text with discussions and suggestions relating to its contents, with critical comments on earlier drafts, and with careful proofreading that uncovered many errors (of which some would have been somewhat embarrassing).

I am particularly indebted to Claus Reinke who gave Chaps. 5 to 8 and Appendix A a very thorough going-over, made some valuable recommendations that helped to improve verbal explanations and also the formal apparatus, specifically in Appendix A which I have largely adopted from his excellent PhD thesis, and provided me with a long list of ambiguities, notational inconsistencies and errors. Some intensive discussions with Sven-Bodo Scholz on head-order reduction, specifically on the problem of shared evaluation, led to substantial improvements of Chaps. 6 to 8. He also pointed out quite a few things in Chaps. 12 and 13 that needed clarification. I also had two enlightening discussions with Henk Barendregt and Rinus Plasmeijer on λ-calculus and on theorem proving which helped to shape Chaps. 4, 11 and Appendix B. Ulrich Bruening checked and made some helpful comments on Chaps. 13 and 14. Hans Langmaack was always available for some insightful discussions of language issues.

Makoto Amamiya gave me the opportunity to teach parts of this text in a one-week seminar course at Kyushu University in Fukuoka/Japan. The ensuing discussions gave me a fairly good idea of how the material would sink in with graduate students who have a slightly different background, and they also helped to correct a few flaws.

Kay Berkling, Claudia Schmittgen and Erich Valkema carefully proofread parts of a text that was more or less unfamiliar scientific territory to them, pointing out a few things that needed to be clarified, explained in more detail (by more examples), or simply corrected.

Last, not least, I wish to thank the people at Springer for their support of this project, especially Ingeborg Mayer, Ronan Nugent, Frank Holzwarth and, most importantly, Douglas Meekison who as a copyeditor did an excellent job of polishing the style of presentation, the layout of the text, and the English. There was hardly anything that escaped his attention.

... and there was Moni whose occasional peptalks kept me going.

Werner Kluge, November 2004

Contents

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.

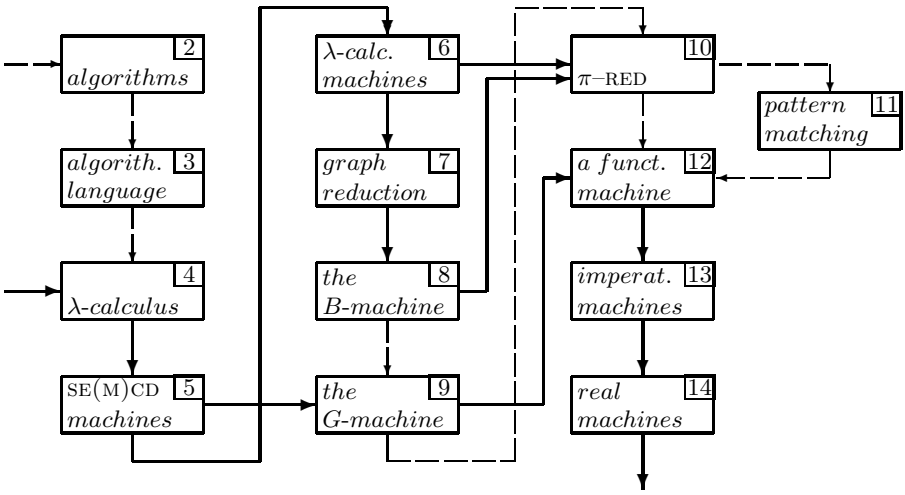
This approach is clearly derived from the programming language C, which knows only flat procedure definitions. Except for references to global variables, these definitions are closed and thus are perfect candidates for direct compilation to code that makes efficient use of the windows.

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.

2

Algorithms and Programs

The art of writing algorithms, also referred to as algorithmics, is undisputably the most important discipline of computer science. Generally speaking, algorithms are recipes that tell us how problem specifications may be transformed step by step into problem solutions, which is exactly what we expect computers to do for us, and what they can do with amazing speed and reliability.

It takes very little to formulate algorithms that can be understood and executed mentally (or with the help of paper and pencil) by human beings who have some moderate mathematical background. All that is needed is a few syntactical constructs (or figures) to specify elementary operations, some constructs by means of which complex operations can be composed from simpler ones, and a finite set of rules that, in an orderly way, transforms these constructs into others until no more rules are applicable, at which point a problem solutions is assumed to have been reached.

We will refer to algorithms as being

- abstract if they are specified using some mathematical notation that merely defines a partial ordering among operations (or rule applications) that reflect the logical structures of the problems at hand, and if no particular mechanisms for performing the operations are assumed;
- concrete or programs if they are specified with execution by machinery in mind, in which case it may be necessary to include detailed work plans (schedules) so that the machine's various gadgets do the right things the right way in the right order.

It is generally possible to translate abstract algorithms into executable programs, and – as we will see later on – we can design abstract or real computing machines that accept and execute abstract algorithms as programs.

Abstract algorithms are usually specified using

- expressions (or terms) that are (recursively) constructed from atomic components such as constant values, variables, primitive arithmetic, logic and relational operators, and of parentheses that define nestings of (sub)expressions in expressions;

- selector expressions that, depending on the value of a predicate or an index, compute just one of two or more alternative subexpressions;
- means for structuring data, e.g., in the form of tuples or lists, and a set of primitive operations to compose complex from simpler structures and to decompose structures into substructures;
- defining equations for abstractions (of variables from complex expressions) that are intended to make the representation of algorithms more concise, giving it more structure if the same computations have to be carried out repeatedly, and possibly in different parts of the algorithm.

We expect meaningful algorithms to be composed of only finitely many legitimate syntactical constructs to which transformation rules may be applied. The rules should be effectively computable, meaning that they can be executed mechanically with reasonable effort. The algorithms should terminate with problem solutions after finitely many rule applications (which cannot be guaranteed in general), and the solutions should be determinate, meaning that there is at most one rule applicable to every syntactical construct, and that problem solutions are invariant against alternative sequences of rule applications.

However, termination and determinacy of results may not necessarily be desirable algorithmic properties. On the one hand, there are algorithms that (hopefully) never terminate but nevertheless do something useful. Well-known examples in computing are the very basic cycle of issuing a prompter, reading a command line, and splitting off (and eventually synchronizing with) a child process for its interpretation, as it is repeatedly executed by a UNIX shell, or, on a larger scale of several interacting algorithms, the operating system kernel as a whole, which must never terminate unless the system is shut down.

On the other hand, there are term rewrite and logic-based systems where the transformation rules are integral parts of the algorithms themselves. Given the freedom of specifying two or more alternative rules for some of the constructs, these algorithms may produce different problem solutions for the same input parameters, depending on the order of rule applications.

When it comes to executing abstract or concrete algorithms by computing machines, there is usually a considerable gap to be bridged between the high-level notation in which these algorithms are specified and the binary code that can be interpreted by the machines. What can be read and understood by human beings is completely indigestible to machines, and what can be processed by machines is, other than within a very small scope of just a few instructions, totally incomprehensible to human beings. It usually takes several levels of representations of algorithms, using notations with increasingly finer resolution of the computational steps that need to be performed, to bridge this gap.

We may think of these levels as a hierarchy of some $n+1$ languages $L_i \mid i \in \{0, \ldots, n\}$, with L_n at the higher and L_0 at the lower (the hardware) end. Each of these languages is executed by another abstract (or real) processor

$P_j \mid j \in \{0, \ldots, n-1\}$ specified in terms of the language of the next lower level. More precisely, the language L_i is executed by a processor P_{i-1} implemented in the language L_{i-1},[1] which may be depicted as

$$L_n \xrightarrow{P_{n-1}} L_{n-1} \cdots \xrightarrow{P_i} L_i \xrightarrow{P_{i-1}} L_{i-1} \cdots \xrightarrow{P_0} L_0 \ .$$

The 'processor' for the language L_0 does not quite fit into this picture since at this level we have a change of paradigm to electronic circuitry and sequences of pulses that make it function.

The processors $P_{i-1} \mid i > 1$ of the higher language levels may be

- either compilers that translate algorithms (or programs) written in the languages L_i as a whole into programs of the language L_{i-1} that have the same meaning (or the same semantics);
- or interpreters written in the language L_{i-1} that execute the constructs of the language L_i one by one.

A typical such hierarchy by which high-level languages are implemented on contemporary computing machines is depicted in Fig. 2.1.

Algorithms (or programs) written in a high-level language are first translated by a compiler frontend rather schematically into some intermediate language that abstracts from the specifics of the underlying machine but allows for some standard code optimizations. Programs of this intermediate language, nowadays typically C, are translated by a compiler backend into assembler code composed of sequences of symbolic machine instructions. This code undergoes various fine-tuning optimizations that take into account particularities of the processor architecture such as the available register set or pipelined instruction execution. The symbolic assembler code thus obtained is then one-to-one transliterated (or assembled) into sequences of binary-coded instructions for interpretation by the machine. This assembler code is the lowest language level accessible to programming. It is used primarily by compiler writers and system programmers who need to implement some performance-sensitive kernel routines, but hardly ever by ordinary application programmers. Below this level, which in fact defines the architecture of the machine as it is described in manuals, there are one or two hard-wired levels of interpreters for machine instructions, of which the lowest one controls the various electronic circuits that finally do the job.

In the following we will be concerned with the upper levels of the language | processor hierarchy, specifically with a variety of abstract state-transforming machines (processors) that either interpret abstract algorithms directly or,

[1] It should be noted here that this is a somewhat idealized picture, insofar as in practice the compiler or interpreter P_{i-1} need not necessarily be written in the language L_{i-1}; any other suitable language may be used for this purpose.

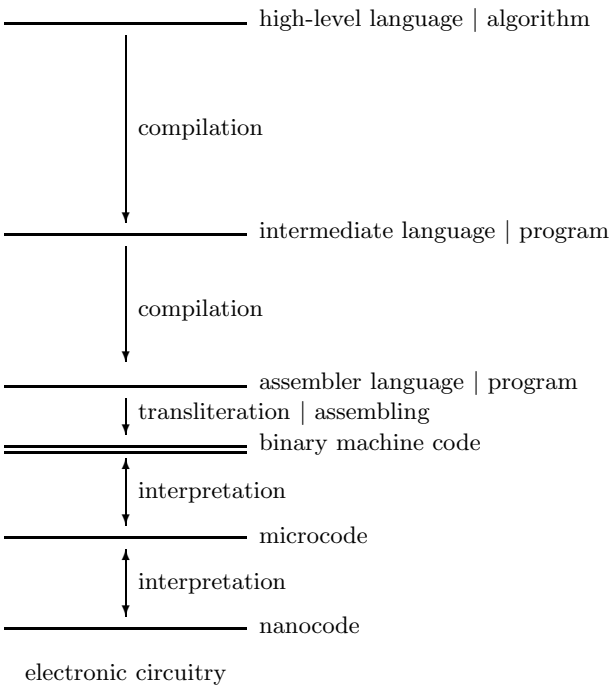


Fig. 2.1. A typical hierarchy of languages and language processors

on a lower level, interpret abstract instruction-based machine code to which abstract algorithms are being compiled. Focusing on abstract machines will help us to identify, in a clean setting, the most essential mechanisms and runtime environments for program execution and how they translate into the architectures and operating principles of conventional computing machines.

2.1 Simple Algorithms

We begin our excursion into algorithmics with some simple examples to introduce in an informal way the very basics of the design of abstract algorithms and of the transformation rules and mechanisms required to execute them.

Designing an algorithm should set out with a formal specification of what is to be computed. This is always advisable as a first step just to make sure that the problem is fully understood, that the domains of input parameters and essential properties of the intermediate and final problem solutions are precisely characterized, and that the solutions actually produced by the algorithm can be checked against these properties.

2.1.1 Getting Started with Some Basics

Our first algorithm is intended to compute the volume of the frustum of a pyramid with a square-shaped base area, as shown in Fig. 2.2. This volume may be computed by subtracting the volume of the smaller pyramid that sits on top and has height h_1 and base size r_1 from the volume of the larger pyramid with height h_2 and base size r_2. With $h = h_2 - h_1$ as the height of the frustum of the pyramid, we get for its volume the formula

$$V_h = (1/3) * h * ((r_1 + r_2)^2 - r_1 * r_2) \ .$$

We also know that this formula is meaningful and yields correct results only if the variables $h,\ r_1,\ r_2$ are placeholders for real numbers greater than zero and the value of r_2 is greater than that of r_1.

Thus, the problem may be formally specified as

$$V_h = (1/3) * h * ((r_1 + r_2)^2 - r_1 * r_2) \ ,$$

where[2]

$$(h,\ r_2,\ r_1 \in I\!R^+) \ \wedge \ (r_2 > r_1) \ .$$

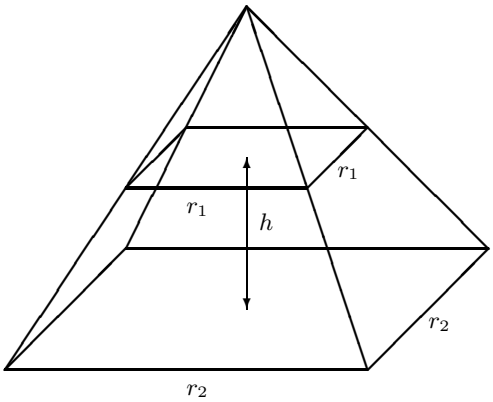


Fig. 2.2. Parameters for computing the volume of a frustum of a pyramid

The right-hand side of the equation for V_h looks very much like an algorithm if we know the rules for evaluating nested arithmetic expressions. However, if we consider the variables as 'unknowns' in a mathematical sense, nothing can be done if they are not replaced by real numbers to which the arithmetic operators $+,\ -,\ *,\ /$ can be applied.[3] The problem that we need

[2] $I\!R^+$ denotes the set of real numbers greater than zero.

[3] We assume here that no symbolic computations can be performed that transform expressions containing variables into other, possibly more concise, expressions.

to solve is to come up with some construct (and associated mechanism) that, in some orderly way, substitutes values for these variables so that the formula can actually be evaluated. But before the evaluation actually takes off, we must make sure that the values substituted for the variables satisfy the logical formula of the where clause underneath the formula for V_h. Otherwise, the algorithm should indicate somehow why the evaluation cannot proceed or is bound to produce an erroneous result.

We will address the problem of substituting values for variables first. All we need to do here is to define a suitable operator that we can apply to these values. The expression V_h must be one part of it, the other part must be the one that substitutes values for variables in this expression. We may give this operator a name (or an identifier), say f, and define it by the equation[4]

$$f \;=\; \texttt{lambda}\; h\; r_1\; r_2\; \texttt{in}\; (((1/3) * h) * ((r_1 + r_2)^2 - (r_1 * r_2))) \;\; .$$

The keyword `lambda` denotes a constructor that is said to bind the variables $h,\ r_1,\ r_2$ in the expression following the keyword `in`. This notation may look a little strange at first sight, but we may obtain a more familiar notation that means the same thing if we move the variables under the `lambda` over to the left-hand side of the equation and simply write

$$f\; h\; r_1\; r_2 \;=\; (((1/3) * h) * ((r_1 + r_2)^2 - (r_1 * r_2))) \;\; .$$

This denotes f as a function, also called an abstraction, of three variables (or formal parameters) $h,\ r_1,\ r_2$. The expression on the right-hand side is said to be the function (or abstraction) body which implements the algorithm that computes function values.

Both notations are semantically equivalent, but in the following we prefer the one that uses the `lambda`-construct because it lends itself more elegantly to describing operational aspects. In fact, we may consider the `lambda`-construct as an operator replacing the identifier f that may be applied to operands, in this particular case to the values (or actual parameters) that we wish to substitute for the formal parameters $h,\ r_1,\ r_2$.

Before we specify such an application and describe how it evaluates, we would like to do something that may look a little unusual but gives an interesting touch to the way we would like to compute. We decide to apply the operator f to numerical values for the variables r_1 and r_2, say 2 and 3, respectively, but if we have not yet made up our mind about what the value for the height h should be, we may simply wish to substitute some nonnumerical dummy value a for it and see how far we can drive the evaluation ahead. We may write this application as

$$(f\;\; a\;\; 2\;\; 3) \;\; ,$$

[4] We use here a fully parenthesized infix notation for primitive binary operations such as $(a + b)$ for adding a and b.

i.e., we put what we consider the operator in the first syntactical position and the operands in some specific order in the subsequent positions inside the parentheses.

Looking at this application as it is, it could not be evaluated to anything else, since it has in operator position a variable that, on its own, merely represents itself. The application therefore would have to be considered a constant expression and simply be left unchanged.

However, since in a larger context we also have a defining equation for f, we can replace it by its right-hand side to obtain

$$(\texttt{lambda}\ h\ r_1\ r_2\ \texttt{in}\ (((1/3) * h) * ((r_1 + r_2)^2 - (r_1 * r_2)))\ \ a\ \ 2\ \ 3)\ \ .$$

Now, this application makes more sense, because it says that the operator `lambda` $h\ r_1\ r_2$ `in` (...) must be applied to the operands $a,\ 2,\ 3$, and the operation to be performed must substitute for occurrences of the variables $h,\ r_1,\ r_2$ in the function body (...) the values $a,\ 2,\ 3$, respectively, assuming that the ordering of operands in the application follows the ordering of formal parameters under the `lambda`. This operation obviously yields the expression

$$(((1/3) * a) * ((2 + 3)^2 - (2 * 3)))\ \ ,$$

which now has numbers in place of the variables r_1 and r_2 and can therefore be evaluated further.

This expression prescribes no particular order for evaluating its subexpressions other than that arithmetic operators can only be applied to numbers and that therefore the evaluation must proceed from innermost to outermost. So, we may evaluate $(1/3)$ to 0.3333..., $(2 + 3)$ to 5 and $(2 * 3)$ to 6 in order to obtain the intermediate expression

$$((0.3333 * a) * (5^2 - 6))\ \ ,$$

in which we have to evaluate 5^2 to 25 first and then subtract 6 from it to get

$$((0.3333 * a) * 19)\ \ .$$

This is all that can be done, since a is not a number, so it cannot be multiplied by 0.3333, i.e., the expression $(0.3333 * a)$ must be left as it is, with the consequence that the multiplication by 19 cannot be done either.

We may consider this result perfectly legitimate: it is an expression that defines a value for the volume of the frustum of a square-shaped pyramid with specific base size values r_1 and r_2 but with an as yet unspecified height, which for the time being is represented by the symbolic value a. However, we also note that this computation was carried out without regard for the logical formula of the formal problem specification, which says, more precisely, that the things substituted for the variables should all be numbers greater than zero, that the value substituted for r_1 should be smaller than r_2, and that the resulting value for the volume should be a number greater than zero too. The

algorithm we have used here did not perform the respective tests, and that is exactly why we succeeded with the evaluation as far as we did.

Including these tests in the algorithm requires one more operator *is_num* that, when applied to something, returns a Boolean value (either `true` or `false`) that tells us whether this something is a number or not, and a selector expression of the form

$$\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \ ,$$

which, depending on the Boolean value to which the subexpression e_0 is assumed to evaluate, returns the value of either e_1 (which is called the consequent) or e_2 (which is called the alternative).

With these tests, the function f looks a lot more complicated but it guarantees that only correct results for correctly specified parameters are returned as function values:

$f\ =\ \texttt{lambda}\ h\ r_1\ r_2\ \texttt{in}$
$\quad\texttt{if}\ (((\texttt{is\_num}\ h)\ \texttt{and}\ (\texttt{is\_num}\ r_1))\ \texttt{and}\ (\texttt{is\_num}\ r_2))$
$\quad\texttt{then if}\ (((h\ \texttt{gt}\ 0)\ \texttt{and}\ (r_1\ \texttt{gt}\ 0))\ \texttt{and}$
$\qquad\qquad\qquad((r_2\ \texttt{gt}\ 0)\ \ \texttt{and}\ (r_2\ \texttt{gt}\ r_1)))$
$\qquad\texttt{then}\ (((1/3) * h) * ((r_1 + r_2)^2 - (r_1 * r_2)))$
$\qquad\texttt{else}$ "parameters out of range"
$\quad\texttt{else}$ "one parameter is not a number" .

If one of the parameters to which the function is applied is not a numerical value, as in the preceding example, or if one of the parameters is a negative number, or if the base size at the top is greater than the base size at the bottom, then the function returns as a value a character string saying why the algorithm could not be executed as intended.

The problem that we have solved here, though it gives us some initial idea of how to design algorithms in general, is rather straightforward and trivial insofar as the number of operations to be performed is fixed irrespective of the actual parameter values. Other than for the first step that, in the application ($f\ a\ 2\ 3$), substitutes the identifier f by the right-hand side of the defining equation, the size of the expression shrinks monotonically with every arithmetic operation performed.

However, the overwhelming majority of computational problems are far from being that simple. They typically involve operations that have to be performed repeatedly, the number of repetitions often depends in intricate ways on the input parameters, e.g., on the sizes of data structures or on the number of alternatives that need to be checked out, and, even worse, the repetitions may cause the computations to continuously expand in space. In fact, such repetitions are the essence of being able to compute everything that is intuitively computable, or computable in principle.

2.1.2 Recursive Functions

A simple example to illustrate this concept is the computation of the product of all integer numbers within the interval $1 \ldots n$, also known as the factorial of n. Given some numerical value $n \in \{1, 2, \ldots\}$, the algorithm must be capable of unfolding an expression of the form

$$n! \;=\; 1 * 2 * 3 * \ldots * i * (i+1) * \ldots * (n-1) * n \;\; .$$

A formal specification of how this algorithm must be designed may be directly obtained from this n-fold product. It tells us that the factorial of 1 is trivially 1, and that the factorial for numbers $n > 1$ may be computed as the product of n and of the factorial of $n-1$, i.e., we have

$$1! \;=\; 1 \;\text{ and }\; n! \;=\; n * (n-1)! \mid n > 1 \;\; .$$

This specification can be directly translated into an algorithmically computable function

$$fac\, n = \texttt{if}\ (n\ \texttt{gt}\ 1)\ \texttt{then}\ (n * (fac\ (n-1)))\ \texttt{else}\ 1$$

or, when the equivalent `lambda`-notation is used,

$$fac = \texttt{lambda}\, n\ \texttt{in}\ \ \texttt{if}\ (n\ \texttt{gt}\ 1)\ \texttt{then}\ (n * (fac\ (n-1)))\ \ \texttt{else}\ 1 \;\; .$$

This function is said to be recursive, since the function identifier fac recurs inside the function body to apply the function to the operand expression $n-1$. This recursion continues as long as the argument of fac remains greater than 1. When applying fac to an initial value $n > 0$, the algorithm is bound to terminate after n recursive calls of fac since n is monotonically decremented by 1 until it is down to 1, in which case the function returns the value 1.

Figure 2.1.2 illustrates how the computation of fac proceeds if we use the simple rules of substituting, whenever needed, function identifiers by the right-hand sides of their defining equations, formal parameters by the operands the functions are applied to, and expressions specifying primitive operations by their values.

The computation sets out with the application $(fac\ 5)$ and, in the first step, substitutes (or expands) the identifier fac by the operator `lambda` n `in`... on the right-hand side of its defining equation. Applying this operator to the operand value 5 yields, in the next step, the function body expression in which all occurrences of the `lambda`-bound variable n are now substituted by 5. This enables the predicate (5 `gt` 1) of the selector expression to be evaluated to `true`, which picks the consequent for further evaluation, and the rest of the expression is dropped. What is now left is an expression that multiplies 5 by the application of fac to 4, in which fac is again expanded as before. These steps are repeated until fac is applied to 1, which returns the value 1, and

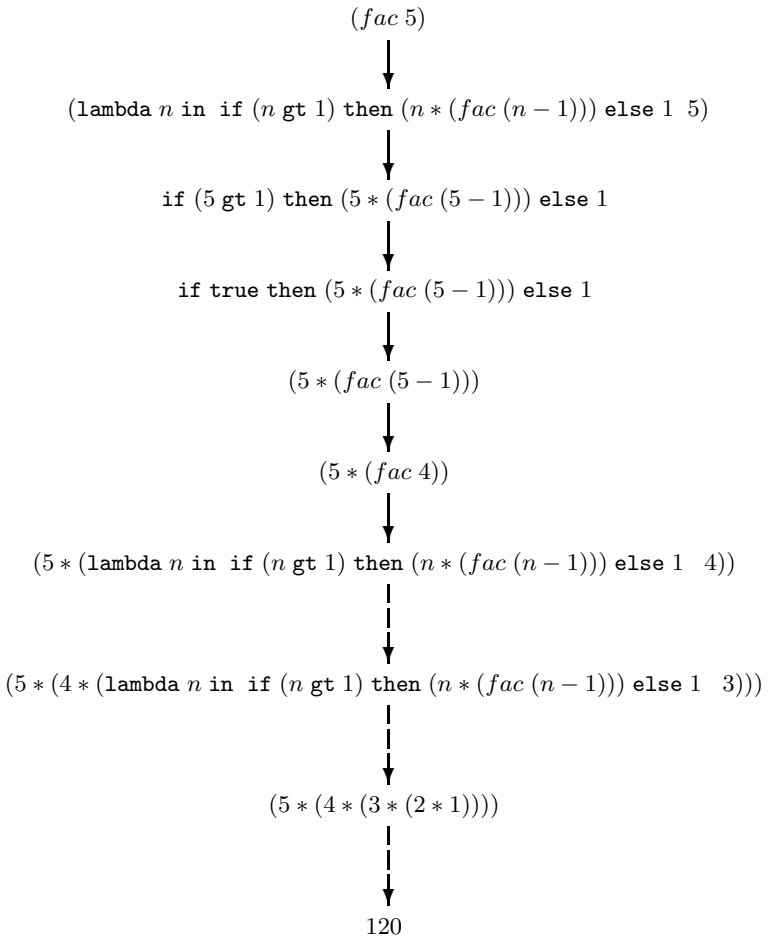


Fig. 2.3. Stepwise execution of the recursive algorithm for the factorial of 5

the expression left is the expanded product of all numbers from the interval $1 \ldots 5$, as in the second last line. From there it takes four multiplications to arrive at the value 120.

We note that the expression periodically expands when the function identifier is replaced by its defining expression and shrinks when this expression is evaluated, and, in doing so, unfolds the multiplication of numbers from 5 down to 1, which subsequently collapses into the result value.

This very regular behavior allows us to give fairly precise figures for the time and space that it takes to perform the computation. It obviously requires n recursive calls of fac to compute the factorial of n. Assuming that it takes some constant time to evaluate the function body (other than in the case $n = 1$, there is each time the same number of primitive operations to be

carried out), we can say that the execution time grows linearly with n. This is usually denoted as $O(n)$, which is to be read as 'the time is of order n (or of complexity $O(n)$)'. A more precise figure would require that we know exactly how much time it takes on a particular machine to perform individual operations, which in turn may depend in intricate ways on the context in which they occur.

A similar estimate can be made for the space demand. We know that in the i-th recursion the size of the expression is given by the number of characters it takes to represent $i-1$ nested multiplications of numbers plus the fully expanded function body. We can now argue that the function body contributes only some character string of constant length that we can safely ignore if n becomes very large, and that we are interested only in the dynamic parts that change with n. This means that we need to worry only about the space that it takes to build up n multiplications, which again grows linearly with n, or is of complexity $O(n)$.

If we wish to compute the product of some n numbers, there is no way around generating those $n-1$ pairwise multiplications, i.e., the complexity in time remains $O(n)$ irrespective of the way we do it. But we can save on space consumption if we design an algorithm that, rather than fully expanding $n-1$ nested multiplications before actually evaluating them, right away multiplies the actual value generated by some function application by an intermediate product accumulated by the preceding function calls.

A formal specification can be derived from the fact that

$$n! \;=\; 1*2*3*\ldots*i*(i+1)*\ldots*(n-1)*n$$

may alternatively be expressed as

$$n! \;=\; 1*2*3*\ldots*i*r_{i+1},\ \text{where}\ r_{i+1} \;=\; (i+1)*\ldots*(n-1)*n\ \ ,$$

i.e., we have

$$r_n \;=\; n \quad \text{and} \quad r_i \;=\; i*r_{i+1} \mid i<n\ \ .$$

This variant of computing factorial of n requires a function fac_it of two formal parameters n and r, the latter of which passes the accumulated intermediate product from one recursive call to the next:

$$fac\_it \;=\; \texttt{lambda}\ r\ n\ \texttt{in if}\ (n\ \texttt{gt}\ 1)\ \texttt{then}\ (fac\_it\ (r*n)\ (n-1))\ \texttt{else}\ \ r$$

Figure 2.4 illustrates some steps of computing the application $(fac\_it\ 1\ 5)$. We note that the sequence of expressions periodically expands and then collapses again to applications of fac_it to accumulated products r in the first operand position and decremented values of n in the second position, until it terminates, again with the value 120. That is to say, the expression never expands beyond the size of the application that has the identifier fac_it substituted by the right-hand side of its defining equation, i.e., for all practical purposes the computation takes place in constant space, or is of space complexity $O(1)$.

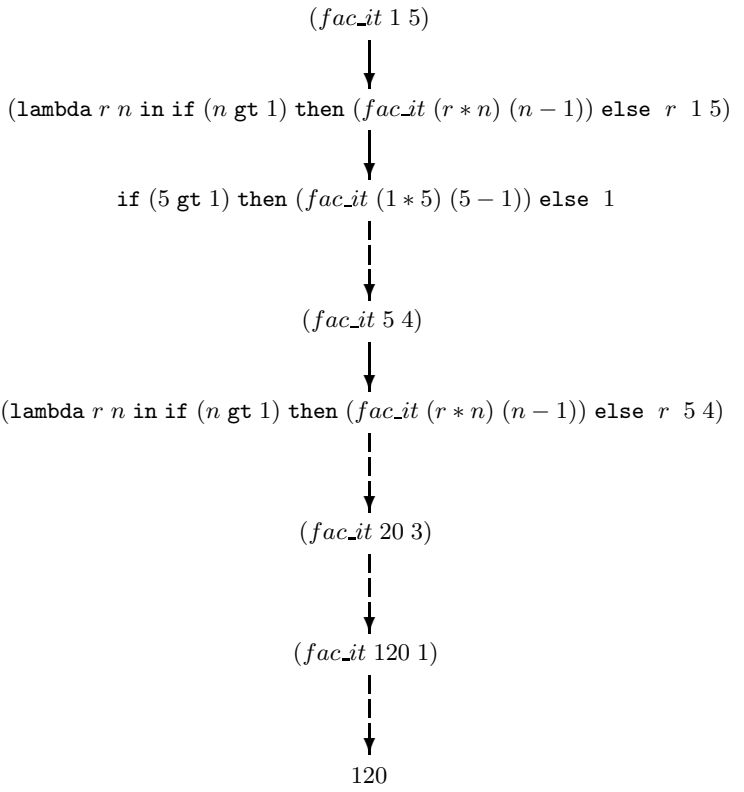


Fig. 2.4. Execution steps of the iterative algorithm for the factorial of 5

Recursive functions that behave like this are said to be tail-recursive because calling the function recursively is always the last action in evaluating current instances of the function body, i.e., nothing is left to be done when the computation returns from the next call. Alternatively, these functions may also be referred to as being iterative since they are equivalent to while statements of conventional programming languages that perform iterations over statement blocks as long as some predicate term evaluates to `true`.

Both fac and fac_it are bound to terminate for numerical values. If n is chosen to be greater than zero then these functions terminate after n recursive calls, otherwise they terminate trivially with the value 1, even if n is a negative number, in which case the result may not be what we really want. What has been omitted here for reasons of simplicity are tests that the actual value of n is greater than zero, and – what seems even more important – that the value substituted for n is a number after all.

However, failing to do the latter need not have fatal consequences: substituting, say, some character string "*aabb*" for n instead of a number would mean that the value of ("*aabb*" `gt` 1) would have to be computed next as the

selector of the `if_then_else` expression. Since it makes no sense to compare a character string with a number, we can decide to leave this expression as it is and subsequently leave the entire `if_then_else` clause untouched since without a Boolean value no selection can be made between the consequent and the alternative. The computation of fac_it would then stop with the expression

$$\texttt{if}\ (\text{"}aabb\text{"}\ \texttt{gt}\ 1)\ \texttt{then}\ (fac\_it\ (1 * \text{"}aabb\text{"})\ (\text{"}aabb\text{"} - 1))\ \texttt{else}\ 1\ \ .$$

and return it as a result. The mistake that has been made would be fairly obvious: a character string and a number are not compatible with respect to the operator `gt` or, for that matter, with respect to primitive arithmetic, logic and relational operators in general. Hence such operations cannot be performed in a meaningful way and, rather than producing some ambiguous error message, the expressions that specify them are simply left as they are, whereupon the person who wrote the algorithm may decide what went wrong.

2.1.3 The Termination Problem

We will now briefly discuss another two algorithms that address the problem of termination.

The first one is the well-known Euclidean algorithm that computes the greatest common denominator of two positive integer numbers. It may be specified using two functions, of which one calls upon the other as a subfunction:

$$\begin{aligned} gcd\ =\ &\texttt{lambda}\ u\ v\ \texttt{in}\ \texttt{if}\ (u\ \texttt{eq}\ v)\\ &\qquad\texttt{then}\ u\\ &\qquad\texttt{else if}\ (u\ \texttt{gt}\ v)\\ &\qquad\qquad\texttt{then}\ (gcd\ v\ (mod\ u\ v))\\ &\qquad\qquad\texttt{else}\ (gcd\ u\ (mod\ v\ u))\ \ , \end{aligned}$$

$$mod\ =\ \texttt{lambda}\ u\ v\ \texttt{in}\ \texttt{if}\ (u\ \texttt{leq}\ v)\ \texttt{then}\ u\ \texttt{else}\ (mod\ (u - v)\ v)\ \ .$$

When the function *gcd* (for greatest common denominator) is applied to actual parameter values greater than zero, it is bound to terminate since both parameters are monotonically decremented by calls of the function *mod* (which computes the modulus of two values) until the termination condition (u `eq` v) is reached, which in the worst case happens when both values come down to 1. However, there is no easy way of telling after how many recursive calls of *gcd* and *mod* this may be the case, other than that this is a number anywhere between 1 and the maximum of the initial values of u and v. Thus, we can only say that the worst-case complexity with regard to time is $O(max(u, v))$. Since both functions are tail-recursive, the computation can be performed in constant space though, i.e., in this respect we have a complexity of $O(1)$.

Moreover, the algorithm does not terminate if both arguments are negative numbers (other than trivially for those that are equal), that are outside

its intended domain. Consider as an example the application $(gcd\ -2\ -1)$. It subsitutes -2 for u and -1 for v in the body of *gcd*, where it recursively calls $(gcd\ -2\ (\ mod\ -1\ -2))$. From here on, the computation becomes trapped in recursive calls of *mod*, as each time the value substituted for u is stepped up by $+2$, whereas the value -2 substituted for v remains unchanged. Thus the predicate $(u\ \texttt{leq}\ v)$ keeps evaluating to `false`, i.e., it never reaches the termination condition. Essentially the same happens with all other combinations of negative arguments. However, this problem can be easily fixed by including in the function definitions tests for both argument values being numbers greater than zero.

The second algorithm does not seem to compute anything useful at all. For good reasons, it is called the roller-coaster algorithm. It is defined by a recursive function f_{rc} of one parameter n that terminates whenever the value of n becomes 1. However, since n is changed by f_{rc} to an even number three times its current value (plus 1) if this value is odd, and divided by two if this value is even, the function values appear to oscillate between high and low numbers, and it is not clear whether the algorithm terminates at all:

$$f_{rc} = \texttt{lambda}\ n\ \texttt{in if}\ (n\ \texttt{eq}\ 1)\ \texttt{then}\ 1 \\ \texttt{else if}\ (\texttt{odd}\ n)\ \texttt{then}\ (f_{rc}\ ((3 * n) + 1))\ \texttt{else}\ (f_{rc}\ (n/2))\ .$$

What can safely be said though is that the algorithm terminates after k recursive calls of f_{rc} if n is chosen to be some value 2^k, where k is an integer number greater than or equal to zero. In that case, the function keeps dividing n by 2 until the value $n = 2^0 = 1$ is reached, at which point it stops and returns 1 as result. For all other values of n, it cannot be decided by formal reasoning whether or not the algorithm comes to a halt or whether it runs forever. The only way to answer this question is to run the algorithm without any time bounds for all values of n, starting with $n = 3$ and proceeding upwards, and see what happens. This has actually been done for all values up to 2^{30} and, interestingly enough, the algorithm has been found to terminate for all of them. This means that, while oscillating up and down, intermediate values of n are obviously bound to hit some value 2^k eventually, from where it takes another k calls of f_{rc} for the algorithm to come to an end.

2.1.4 Symbolic Computations

Another interesting aspect of designing algorithms concerns symbolic simplifications prior to computing, say, actual numbers. The savings, in terms of computational steps performed, may be substantial if expensive function calls can be avoided by substituting functions directly inside each other and evaluating them as far as possible symbolically, without having certain function parameters instantiated with actual values.

Consider as an example the computation of function values by means of a Taylor series that for functions $f(u)$ is generally of the form

$$f(u) \;=\; f^{(0)}(0) + \frac{f^{(1)}(0)}{1!} * u + \frac{f^{(2)}(0)}{2!} * u^2 + \ldots + \frac{f^{(n)}(0)}{n!} * u^n \;\;,$$

where $f^{(n)}(0)$ denotes the nth derivative of f at point $u = 0$.

If in a given algorithm we have to compute Taylor series for different functions and several times for each function, it might just be a good idea to implement a scheme for generating such series that separates the generation of its terms from the computation of its coefficients on the one hand and from the computation of the powers of u on the other hand. To this end, we may define a function

$$\begin{aligned} taylor = {} & \texttt{lambda}\ i\ n\ u\ \texttt{in} \\ & \texttt{if}\ (i\ \texttt{eq}\ n)\ \texttt{then}\ ((coef\ i) * (pow\ i\ u)) \\ & \texttt{else}\ (((coef\ i) * (pow\ i\ u)) + (taylor\ (i+1)\ n\ u)) \end{aligned}$$

that, when applied to initial values $i = 0$ and some n greater than zero, and where the parameter u is left unspecified, generates the sum over n terms $((coef\ i) * (pow\ i\ u))$. The variables *coef* and *pow* stand for as yet undefined functions that are to compute specific coefficients and powers of u, respectively.

For a Taylor series that computes the *sine*, these functions would have to be specified thus:

$$coef \;=\; \texttt{lambda}\ i\ \texttt{in}\ ((power\ -1\ i)\ /\ (fac((2 * i) + 1)))\;\;,$$

$$pow \;=\; \texttt{lambda}\ i\ u\ \texttt{in}\ (power\ u\ ((2 * i) + 1))\;\;.$$

Here again we use two undefined functions, of which *power* is intended to take its first argument and raise it to the power of its second argument, and fac is intended to compute the factorial of its argument. If we just let them stand as variables and apply the function *taylor* partially as (*taylor* 0 3), then we can expect, as a result, an abstraction in one parameter u that has the computation of *sine* expanded to four terms:

$$\begin{aligned} & \texttt{lambda}\ u\ \texttt{in}\ ((((power\ -1\ 0)\ /\ (fac\ 1)) * (power\ u\ 1)) + \\ & \qquad ((((power\ -1\ 1)\ /\ (fac\ 3)) * (power\ u\ 3)) + \\ & \qquad\quad ((((\ power\ -1\ 2\ )\ /\ (\ fac\ 5\ )) * (power\ u\ 5)) + \\ & \qquad\qquad (((power\ -1\ 3)\ /\ (fac\ 7)) * (power\ u\ 7))))) \;\;. \end{aligned}$$

If *power* is defined as

$$power \;=\; \texttt{lambda}\ v\ j\ \texttt{in}\ \texttt{if}\ (j = 0)\ \texttt{then}\ 1\ \texttt{else}\ (v * (power\ v\ (j-1))$$

and fac is defined as earlier in this section, the above abstraction can be further simplified to:

$$\texttt{lambda}\ u\ \texttt{in}\ (u + ((-1/6) * (u * (u * u)) + ((1/120) * (u * (u * (u * (u * u))))) + ((-1/5040) * (u * (u * (u * (u * (u * (u * u))))))))) \ .$$

Now we have arrived at an abstraction for the Taylor series of *sine*, from which all other function calls have been systematically eliminated, and everything has been evaluated as far as is possible without knowing actual values for the parameter u.[5] Repeated computations of *sine* have thus been reduced to passing one argument value and to performing a total of three additions and fifteen multiplications. The costs of doing the function calls that generate this abstraction must be paid for only once.

The steps that are crucial for this simplification are a partial application of *taylor* that leaves the parameter u uninstantiated, and evaluation of the partially instantiated abstraction body.

Such function specializations may also be obtained when computing new functions from existing ones by application of functions to operands that are themselves functions.[6]

As an example, consider the functions

$$\begin{aligned} twice &= \texttt{lambda}\ f\ u\ \texttt{in}\ (f\ (f\ u)) \ , \\ square &= \texttt{lambda}\ v\ \texttt{in}\ (v * v) \ , \end{aligned}$$

of which *twice* applies its first parameter f, which is assumed to be a function of one parameter, twice to its second parameter u, and *square* computes the square of its parameter v. When applying *twice* to *square* and to the value 2, we obviously obtain as value the square of the square of 2, which is 16.

The sequence of transformation steps that computes this value is shown in Fig. 2.5. The first step of this sequence expands *twice* by the right-hand side of its defining equation, followed by the substitution of *square* for f and of the value 2 for u in the body of this abstraction. In the subsequent steps, the two occurrences of *square* are expanded by the right-hand side of its defining equation, whereupon the innermost application computes the square of 2, returning 4, and the outermost application computes the square of 4, returning 16.

Nothing unusual has been done here other than that we have taken the liberty of substituting an operand that happens to be an abstraction into the body of another abstraction. The abstraction *twice* is applied to two operands, and inside it the abstraction *square* is applied each time to one operand, i.e., we have in all cases what may be called full applications, and everything works out just as expected.

We can now take this one step further and decide that we want to compute from *twice* and *square* a new function of one parameter that double-squares

[5] The coefficients have been left as fractions for reasons of clarity but could of course be evaluated as well.

[6] In fact, we could have done the same with the computation of the Taylor series as well if we had had more sophisticated constructs to hand that, for instance, would allow us to define functions that are local to others.

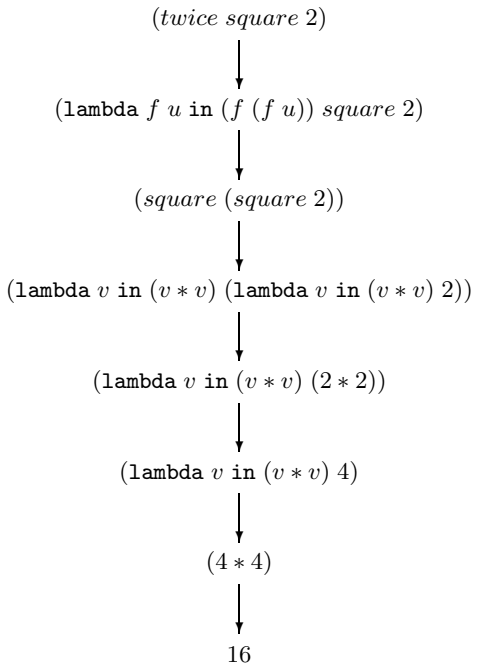


Fig. 2.5. Computing the square of the square of 2

whatever operand value comes along. We can try to do so by applying *twice* just to *square*, in which case we have a partial application of *twice*, and proceed as illustrated in Fig. 2.6.

The interesting transformation step here is the second one, in which the abstraction of two parameters `lambda` f u `in` ... is applied to *square*, with the second operand missing. As the parameters of the abstraction are substituted (and thereby consumed) from left to right, we obtain as a result an abstraction in one parameter, u, in the body of which we have now *square* in the two places of the first parameter f. The next step simply expands both occurrences of *square* by the right-hand side of its defining equation, as a consequence of which another two transformation | substitution steps may be performed to arrive at the abstraction `lambda` u `in` $((u * u) * (u * u))$ that, as expected, multiplies its operand by itself four times.

Since partial applications seem to work as well, we may feel encouraged to try another one. This time we decide to apply the function *twice* partially to itself, hoping to obtain as a result a function *double_twice* of two parameters that applies a function, to be substituted for its first parameter, four times to the operand substituted for its second parameter.

Figure 2.7 shows what happens if we proceed in what appears to be the same way as in the case of computing *double_square* (compare Fig. 2.6).

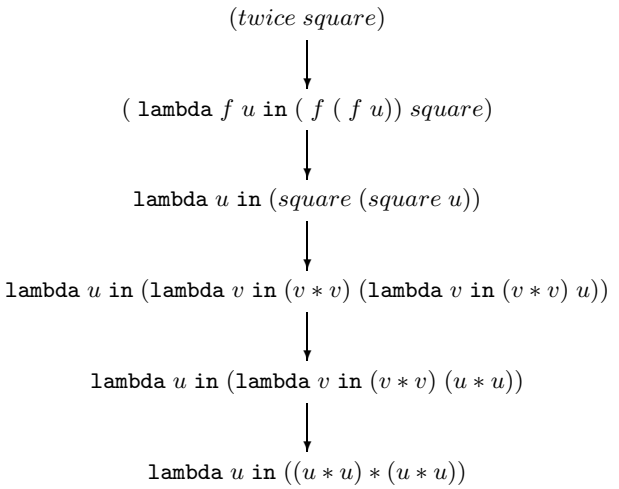


Fig. 2.6. Computing a new function *double_square*

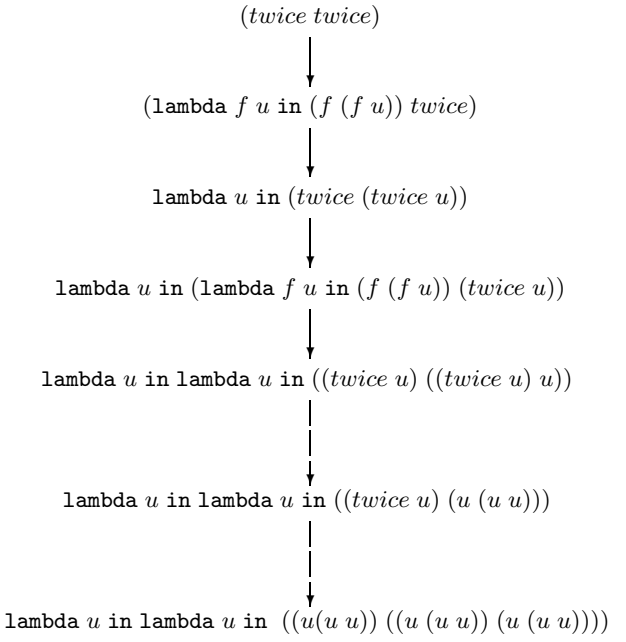


Fig. 2.7. An attempt to compute *double_twice* naively

Again, we do not seem to be doing anything out of the ordinary, we just replace occurrences of *twice* by the right-hand side of its defining equation,

and do the substitutions of `lambda`-bound parameters the same way as before. Nevertheless, something must be going wrong here, because we obviously do not get what we expect, [7], which would be an abstraction of the form

$$\texttt{lambda}\ z\ \texttt{in}\ \texttt{lambda}\ u\ \texttt{in}\ (z\ (z\ (z\ (z\ u))))$$

(where the variable z can be freely chosen, and the choice does not really matter).[8] When applying this abstraction as

$$((\texttt{lambda}\ z\ \texttt{in}\ \texttt{lambda}\ u\ \texttt{in}\ (z\ (z\ (z\ (z\ u))))\ \mathit{square})\ 2)$$

we would get, as desired, *square* applied four times to 2:

$$(\mathit{square}\ (\mathit{square}\ (\mathit{square}\ (\mathit{square}\ 2))))\ .$$

Instead, we obtain as a result of our transformation sequence some strange abstraction of two parameters that are both named u, and an abstraction body that features weirdly nested applications composed of just this variable u which obviously do not make much sense.

Moreover, if we were to apply this abstraction as

$$((\texttt{lambda}\ u\ \texttt{in}\ \texttt{lambda}\ u\ \texttt{in}\ (\ldots)\ \ \mathit{square})\ 2)\ ,$$

then all occurrences of u in the abstraction body would be substituted either by *square* or by 2, depending on whether they are bound by the inner or by the outer `lambda` u, which so far we do not really know yet. All we can say is that neither substitution is correct.

The cause of the problem is obviously an unresolved naming conflict. The accident happens in the fourth step of Fig. 2.7. Here we apply, in the body of the outer abstraction `lambda` u `in` $(\ldots)$, the inner abstraction `lambda` $f\ u$ `in` $(f\ (f\ u))$ to the operand $(\mathit{twice}\ u)$, in which the u is bound to the outer `lambda` u. This operand is substituted for occurrences of f in the abstraction body, with the inner `lambda` u remaining in front. It now parasitically binds the u in the substituted expression $(\mathit{twice}\ u)$, i.e., the u changes its binding status and thus becomes confused with the occurrences of the u that were originally bound to the inner `lambda` u.

As such naming conflicts can potentially occur when substituting, under abstractions, variables that are either free-floating or bound in larger contexts, we have to be careful about these situations. If a conflict does indeed occur, we obviously have to rename one of the conflicting variables to resolve it.

We will return to this naming problem after we have learned a little more about the underlying theory. For now it may suffice to know that such a problem exists.

[7] It should also be noted that we have chosen to do the substitutions in a sequential order that yields the particular nesting of parentheses in the resulting abstraction body. Another sequence of substitutions would result in another structure, which is another indication that we are obviously making a mistake here.

[8] Alternatively, we could have renamed the variable bound by the inner `lambda`, obtaining `lambda` u `in` `lambda` z `in` $(u\ (u\ (u\ (u\ z))))$.

2.1.5 Operating on Lists

Our excursion into algorithmics would be incomplete without introducing at least the very basics of operating on structured objects, which is what is involved in almost all nontrivial computations. The simplest and most versatile forms for representing such objects are lists or linearly ordered sequences of expressions, denoted as $< e_1 \ \ldots \ e_i \ \ldots \ e_n >$. The expressions $e_1 \ \ldots \ e_n$ are called the elements (or components) of the list. A list without any elements is said to be empty, denoted as $<>$.

Lists may be considered expressions themselves, i.e., they may be recursively nested inside each other to construct lists of lists without any bounds.

It takes only three primitive structuring operators to define recursive functions that rearrange (sort) the elements of a list, search lists for elements that satisfy specific properties, decompose lists and construct lists from sublists, etc. These primitives are

- `first`, which, when applied to a nonempty list, returns its first element;
- `rest`, which, when applied to a nonempty list, returns the list without its first element;
- `append`, which, when applied to a first and a second list, returns a new list containing the elements of the first list followed by the elements of the second list;

and are undefined otherwise.[9]

We also need some primitives that test for elementary list properties. These primitives include

- `is_list`, which returns the Boolean value `true` when applied to a list, and `false` when applied to something else;
- `empty`, which, when applied to an empty list, returns the Boolean value `true`, when applied to a nonempty list returns `false`, and is undefined otherwise.

The use of these primitives may be captured in a nutshell by means of an algorithm that reverses the order of elements in a list:

```
reverse = lambda list in
            if (is_list list)
            then if (empty list) then list
                 else (append (reverse (rest list)) < (first list) >)
            else "the argument is not a list"
```

$(reverse \ < e_1 \ \ldots \ e_n > <>)$.

[9] It should be remembered here that we impose no restrictions whatsoever on the expressions that the primitives may be applied to, i.e., the arguments could be something other than lists.

The function *reverse* takes as the parameter *list* a list whose elements are to be rearranged in reverse order. As long as the list substituted for *list* is not empty, the function calls itself recursively with the current rest of *list*, and with its first element appended to it. The algorithm thus terminates with the reverse list returned as the function value after as many recursive calls of *reverse* as there are elements in the list.

2.2 A Word on Typing

Throughout the preceding section we have used, somewhat sloppily, the term 'function' whenever we meant to refer to a particular algorithmic realization of a function, of which – as the computation of factorial exemplifies – there is usually more than one. The mathematical notion of a function, however, is simply that of a mapping of a well-defined set of input values, usually called the function's domain, to a well-defined set of output values, called the function's range. Such sets are, for instance, all nonnegative integer numbers $\{0,\ 1,\ 2, \ldots\}$, the integer numbers in a certain interval, say $\{-1000, \ldots,\ -1,\ 0,\ +1,\ +999\}$, real numbers (which have a fractional part), Boolean values, or character strings, but also sets of composite objects whose components are from any of the other possible sets, such as lists of integers, lists of lists of integers, etc.

This notion of a function enters our algorithmic specifications more or less through the back door in the form of test operators such as `is_num` or `is_list`. They are to make sure that the input parameters are in an intended domain for which the algorithm can somehow be guaranteed to produce results that are within an intended range. It may occasionally even be necessary to further restrict the domain of input values to a certain subset by means of additional predicates.

As escape hatches for tests that fail we have chosen as return values character strings such as "the argument is not a number" which may be considered error messages that are usually outside the intended range of the function.

If we forgo these tests, as we have actually done in some of the algorithms, nothing goes wrong as long as the argument values are within the intended domains, but we may end up either with perfectly valid but undesired results (as in the case of the factorial function applied to negative numbers), with expressions that are not fully evaluated (e.g., when applying fac to character strings), or with nontermination (e.g, when trying to compute the greatest common denominator of negative numbers) otherwise.

Keeping such tests optional means that the responsibility for specifying algorithms that do exatly what they are supposed to do in terms of mapping legitimate input values to correct output values and for intercepting illegitimate inputs or other exceptional conditions lies fully in the hands of the individuals who design these algorithms.

This freedom in designing algorithms contrasts with the notions of types and of type systems supported by almost all programming languages known

today. These notions make functions (or other forms of abstractions such as procedures) as mappings from domain to range sets an integral part of program design. Types are just programming terminology for sets of legitimate values of some kind, including mappings, and type systems specify the rules by which types must be assigned to individual program components in order to have the entire program consistently typed. To this end, the type system of a modern programming language usually comes with a set of basic types for constant values (such as integers, reals, characters, etc.) and for primitive operators, with some composite types for structured data (such as vectors, arrays, records, etc.), and with mechanisms that allow the programmer to define types of his/her own, based on the types provided by the type system.

The idea of a type system is to admit for execution only programs that, beyond being correctly constructed, are also consistently typed, meaning that there are no type conflicts of the kind where, say, a producing operation delivers a result of a type different from that expected by a consuming operation. Such programs are said to be well typed.

Other than raising confidence in the orderly behavior of programs, type consistency is also something that the underlying machine demands. As all contemporary computing systems just execute free-running code composed of sequences of instructions that simply expect the right things, in the form of otherwise indistinguishable bit patterns, to be in the right places in memory at the right time and in the right format, typing plays an important role, as part of the compilation process, in preparing memory layouts and in selecting the appropriate type-specific instructions that operate correctly on them.

Consistent typing may either be inferred from the types assigned by the system to (composite) constant values and primitive operators, which is generally the case with functional languages, or types may have to be explicitly declared in programs, which is what most conventional (imperative) languages demand.

To illustrate how type inference basically works, we consider again as an example the factorial function of subsection 2.1.2:

$$fac \;=\; \texttt{lambda}\; n\; \texttt{in}\; \texttt{if}\; (n\; \texttt{gt}\; 1)\; \texttt{then}\; (n * (fac\; (n-1)))\; \texttt{else}\; 1\;\; .$$

With this function as it is, we may infer a type for fac and for n by the following consideration: the arithmetic operators '$-$' and '$*$' are, by definition of the type system chosen, both assumed to be of the function type $num * num \rightarrow num$ which, loosely speaking, maps pairs of numbers to numbers (with num denoting the type number). For the sake of simplicity, we will assume that the relational operator `gt` is of type $num * num \rightarrow bool$ only, i.e., it maps pairs of numbers to Boolean values. With this in mind, we can immediately infer from the subexpressions $(n-1)$ and $(n\; \texttt{gt}\; 1)$ that the values substituted for n must be of type num in order to be type-compatible with '$-$' and '`gt`', respectively. Looking next at the subexpression $(n * (fac\; (n-1)))$, we can conclude that fac must have the function type $num \rightarrow num$ since it must map arguments of type num (the values of $(n-1)$) to result types num

that must be compatible with the argument type of the operator '$*$'. Now we see that the types of both the consequent and the alternative (which is the constant value 1) of the `if_then_else` clause are *num*, i.e., the type of the entire clause is *num*.[10] This in turn leads us to conclude that the abstraction on the right-hand side of the above equation must have the function type $num \rightarrow num$ since it maps arguments of type *num* substituted for n to function values of type *num*. This is consistent with the type of fac on the left-hand side that has already been inferred.

Thus, applications of fac to numbers are guaranteed to return numbers as results. The only problem left is the type *num* itself. If it is confined to integer numbers greater than zero, then all values returned by the function are also correct; if it includes negative numbers as well, then typing alone does not prevent results that are incorrect with respect to the definition of factorial.

Type inference as outlined above can of course be automated and performed prior to actually executing a program. Ideally, the programmer need not worry much about types unless the program may be rejected because of type inconsistencies, or type inference cannot be completed owing to some ambiguities, in which cases the type system must be helped with type annotations to the program parts that cause those ambiguities.

The less sophisticated alternative to type inference is type annotations to all program variables, which the type system must check only for consistency. This places more responsibility on the programmer but also helps to identify potential causes of type inconsistencies more easily.

Type annotations in the case of the factorial function would typically take the form

$$fac : num \rightarrow num = \texttt{lambda}\ n\ \texttt{in}\\ \texttt{if}\ (n\ \texttt{gt}\ 1)\ \texttt{then}\ (n * (fac\ (n-1)))\ \texttt{else}\ 1\ ,$$

which has the function symbol annotated with the function type, or

$$fac : num = \texttt{lambda}\ n : num\ \texttt{in}\\ \texttt{if}\ (n\ \texttt{gt}\ 1)\ \texttt{then}\ (n * (fac\ (n-1)))\ \texttt{else}\ 1\ ,$$

which has the binding occurrence of n on the right-hand side annotated with the domain type *num* and the function symbol fac on the left-hand side annotated with the range type *num*.

Programming languages which demand that their programs be consistently typed before they can be executed are said to be statically typed, and the type

[10] More precisely, we should consider the `if_then_else` clause as an application of a function `if` to predicate, consequent and alternative as arguments that, depending on the value of the predicate, selects either of the latter two as the result, i.e., it realizes a mapping $bool * type * type \rightarrow type$, where *type* denotes some arbitrary type that is *num* in our particular example.

systems that validate consistent typing either by inference or by checks against type annotations are said to be static type systems.

The idea of dynamically typed languages is that programs may be written with little concern for types, as type checking is done completely at runtime and kept to a minimum. The type-checking mechanisms are assumed to be built into the machines, on the basis of type tags carried along with the objects of the language, and usually intercept only type inconsistencies between primitive operators and the operands they are actually applied to, for instance between arithmetic operators and operands other than numbers. Such languages are also referred to as being untyped or type-free. As we will see later on, they provide more freedom in program design, particularly in the area of symbolic computations, since they permit programming techniques such as self-applications of functions, for instance as in our *double_twice* example, or differently typed consequences and alternatives of `if_then_else` clauses, which would not be accepted by static type systems.

This kind of language is the one we have introduced more or less ad hoc in the preceding section to discuss a few simple algorithms and how they ought to be executed. Expressions with type inconsistencies, as we have indicated, are simply considered constant here and remain unchanged, in this form making explicit what may have gone wrong. The flavor of this approach, as opposed to system-generated error messages, is that it is left to the programmer, not to the system, to decide how to interpret such situations and what to do about them.

As a language of reference, we will therefore settle, in the remaining text, for a dynamically typed (or type-free) algorithmic language that builds on the language fragments of the preceding section, and will add typing only when and to the extent necessary.

2.3 Summary

In this chapter, we have discussed in a rather informal style some important aspects of the prpocess of designing and mentally or mechanically executing algorithms. These aspects include the concepts of abstractions, variables, recursion (and related to it termination problems), symbolic computations and the basics of operating on structured data. We have also given an overview of types and type systems and the role they play in ensuring that algorithms (or programs) compute legimitate output values from legitimate input values.

The chapter is intended as an introduction to and motivation for a more formal treatment of the syntax and semantics of an algorithmic language and of the underlying theory of the λ-calculus, which are the subjects of the next two chapters. The λ-calculus and, to a lesser extent, the particularities of the algorithmic language provide the guidelines for the design of the various abstract computing machines that we are going to study in subsequent chapters.

References

Standard textbooks for an expression-oriented (or function-based) and type-free programming style as advocated in this chapter are primarily those by Abelson and Sussman [AS85] and by Friedman, Wand and Haynes [FWH92], both of which use SCHEME as the language of choice. Programming with the typed function-based language SML is the subject of textbooks by Paulson [Pau96], Ullman [Ull98], and by Hansen and Rischel [HaRi99], and popular texts on purely functional programming are those by Henderson [Hen80], by Bird and Wadler [BiWa88], by Thompson [Tho96], and by Bird [Bird98]. Of a more general nature is an excellent, lucidly written book by Harel on algorithmics [Har92]. It gives an easy-to-comprehend guided tour through algorithmic methods, correctness and efficiency, and the fundamental problems of (in)tractability and (non)computabilty.

3

An Algorithmic Language

Now that we have a fairly good idea of how abstract algorithms should be specified and executed, we can get a bit more formal and introduce a dynamically typed algorithmic language called AL that enables us to precisely express computational problems in a form very similar to the way in which we have been doing this in the preceding chapter. This language must be complete in the sense that all intuitively computable problems can be specified by finite means. It must feature a syntax that defines a set of precise rules for the systematic construction of complex algorithms from simpler parts and a semantics that defines the meaning of algorithms, i.e., what exactly they are supposed to compute, and at least to some extent also how these computations need to be carried out conceptually.

We will call this language expression-oriented since the algorithms are composed of expressions and are expressions themselves, and the objective of executing algorithms is to compute the values of expressions.[1] These computations are realized using a fixed set of transformation rules. The systematic application of these rules is intended to transform expressions step by step into others until no more rules are applicable. The expressions thus obtained are the values we are looking for.

It should be noted here that the term value is given a more general interpretation than is traditionally the case. Values are not necessarily atomic, such as numbers, but may also be rather complex (aggregate) expressions that cannot be transformed into anything else and are therefore in some sense constant. Such constant expressions may include functions, function applications and variables in specific contexts.

Denoting expressions by e or e_i (with the index i assuming values from the set $\{0, \ldots, n\}$), we can describe such a computation as a sequence

$$e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_i \rightarrow e_{i+1} \rightarrow \ldots \rightarrow e_n$$

[1] Though there are close similarities, we have deliberately chosen not to call AL a functional language, since we will use it in a broader sense than what is usually understood by this term.

where e_0 is the start expression, e_n is the terminal expression (or the value of e_0), and the transition from e_i to e_{i+1} comes about by application of a single transformation rule.

The expressions in this sequence have a very important property: if e_n is assumed to be the value of e_0, then we can also say that both e_0 and e_n mean the same thing, or have the same meaning (or semantics), even though syntactically the two expressions may look quite different. This being the case, we can also say that e_1 and, for that matter, all other expressions in the sequence have the same meaning as e_0, or that all expressions in the sequence mean the same thing. The rules by which these expressions are transformed into each other are necessarily meaning-preserving as they obviously replace equals by equals.

This idea may be illustrated by evaluating, in three transformation steps, a simple arithmetic expression:

$$((5+3)*(8/4)) \rightarrow (8*(8/4)) \rightarrow (8*2) \rightarrow 16 \ .$$

Here we see clearly that all (sub)expressions are, by rules of the arithmetic calculus, systematically replaced, from innermost to outermost, by numbers. All expressions in the sequence differ syntactically but have the same meaning, or are semantically equivalent.

It is important to note that we are talking about semantic equivalence here, or about two expressions having the same meaning. This in no way explains what an expression by itself means. In the given language, it is just a syntactical figure, and it may take another language to give it a meaningful interpretation that, in turn, requires yet another level of interpretation, and so on.

So, strictly speaking, we can talk only about syntactical figures and about purely syntactical transformations of expressions into expressions that we consider to be meaning-preserving, but we cannot talk about meaning or semantics itself.

3.1 The Syntax of AL Expressions

We are now going to define the construction of AL expressions from the bottom up, starting with the simplest (the atomic) expressions and moving on to the more complex ones.

The atomic expressions include constant values such as numbers (for the time being we do not distinguish between integers and reals), character strings, the Boolean values `true` and `false`, variables, and the usual symbols for primitive arithmetic, logical and relational operators. These expressions are called atoms (or ground terms) since they are not composed of other expressions of the language.

With the atoms as a basis, we can now proceed to define some syntactical constructs or syntactical forms that are composed of expressions and

are expressions themselves. They may be used to construct more complex expressions by substituting expressions in syntactical positions reserved for expressions. For the definition of these forms we assume that e_0, $e_1, \ldots,$ e_n and e denote legitimate expressions of the language, and that v_1, $v_2, \ldots,$ v_m and $f_1, \ldots,$ f_k denote variables (or identifiers).

In the following itemization, we do not just give the syntax of these forms but also explain very briefly what they are good for. A more formal definition of their semantics is postponed until later in this chapter.

- $(e_0\ e_1 \ldots e_n)$ denotes the application of an expression e_0 to the expressions $e_1 \ldots e_n$. The expression e_0 is said to be in operator position and the expressions $e_1 \ldots e_n$ are in operand positions of the application. Applications are the most important expressions of the language as they are the ones to which transformation rules may be applied. This is possible whenever the expressions in operator position specify legitimate operators that can be applied in some meaningful way to the operand expressions, resulting in other expressions that replace the applications.[2]
- `if` e_0 `then` e_1 `else` e_2 is a special syntactical form that denotes the application of a predicate e_0 to consequent and alternative expressions e_1 and e_2, respectively. It is meant to select either e_1 or e_2 for further evaluation, depending on the value of the predicate e_0 (which must either be `true` or `false`).
- `lambda` $v_1 \ldots v_n$ `in` e_0 denotes an abstraction of the variables $v_1 \ldots v_n$ from the expression e_0, or a nameless function of n formal parameters $v_1 \ldots v_n$ whose body expression e_0 specifies the computation of function values. The `lambda` is said to bind the variables $v_1 \ldots v_n$ in the body e_0, which is also referred to as the scope of what we call the abstractor `lambda` from now on.

  Abstractions are legitimate operators. Applications with abstractions in operator position have a transformation rule defined for them that, loosely speaking, returns the abstraction bodies with occurrences of all `lambda`-bound variables substituted by the expressions that are in operand positions.
- $< e_1\ \ldots\ e_n >$ denotes an n-ary `list` (or a sequence of expressions) as a means for arranging expressions in a particular order, of which it makes sense to talk about a first, a last or some i-th component. Lists may be empty, denoted as $<>$.
- `letrec` $f_1 = e_1\ \ldots\ f_k = e_k$ `in` e_0 is the most complex expression. The keyword `letrec` precedes a set of defining equations that equate the variables $f_1, \ldots, f_k$ with the expressions $e_1, \ldots, e_k$, respectively,

[2] This prefix notation for applications, which requires that operators must always precede operands, is, for reasons of uniformity, used throughout the rest of this text. This includes applications of primitive binary operators that are usually written in infix notation; i.e., something like $(e_2\ prim\_op\ e_1)$ must be rewritten as $(prim\_op\ e_1\ e_2)$.

and binds them in each of the expressions $e_1, \ldots, e_k$ and also in e_0. In fact, these variables may be considered names or function identifiers by which the respective expressions on the right-hand sides may be referenced somewhere else in the entire `letrec` expression.
The idea is to have $e_1, \ldots, e_k$ specified as abstractions that may call each other in a mutually recursive manner by their assigned names (or identifiers). For instance, in the body of the abstraction named f_i we may have occurrences of f_i, in which case the abstraction recursively calls itself, or occurrences of variables $f_j \mid j \neq i$, in which case some other abstractions specified under the `letrec` construct are called.
The value of the entire `letrec` expression is the value of the goal expression e_0, in which occurrences of the identifiers $f_1, \ldots, f_k$ are (recursively) substituted by the right-hand sides of the respective defining equations.

- `let` $v_1 = e_1 \ldots v_n = e_n$ `in` e_0 is a special variant of the `letrec` expression that binds occurrences of the variables $v_1, \ldots, v_n$ nonrecursively just in the goal expression e_0. The value of a `let` expression, again, is defined as the value of e_0 in which all occurrences of the `let`-bound variables are substituted by the values of the expressions on the right-hand sides of their defining equations.
- nothing else is a legitimate expression of the language.

A concise definition of these syntactical forms may be written as

$$
\begin{aligned}
e =_s\ & \mathit{const} \mid \mathit{var} \mid \mathit{prim\_op} \mid \\
& (e_0\ e_1\ \ldots\ e_n) \mid \\
& \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& \texttt{let}\ v_1 = e_1\ \ldots\ v_n = e_n\ \texttt{in}\ e_0 \mid \\
& \texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0 \mid \\
& <> \mid < e_1\ \ldots\ e_n > \mid \\
& \texttt{letrec}\ f_1 = e_1\ \ldots\ f_n = e_n\ \texttt{in}\ e_0\ .
\end{aligned}
$$

An expression e may assume any of the alternative syntactical forms listed on the right of the $=_s$ sign (which denotes syntactical equivalence), and the expression symbols e_i appearing in these forms may be recursively expanded by any of the alternatives. The expansion stops when all expression symbols e_i have been substituted by any of the atoms *const*, *var*, *prim_op* of the language, which denote constant values, variables, and primitive operators, respectively.

Constants are either numbers represented as strings of decimal digits that may include a decimal point, plain text represented as strings of characters enclosed in quotation marks ", or the Boolean values `true` and `false`. Variables are strings of letters, digits and underscores that have a letter as the first character, and the primitive operators include $+$, $-$, $*$, $/$ for arithmetic, `and`, `not`, ... for logical, and `eq`, `neq`, `gt`, ... for relational operations. There are also some primitives such as `first`, `rest`, ..., `append` that operate on lists.

This syntax provides complete freedom for inserting syntactical forms into each other. Any legitimate expression may appear in any syntactical position reserved for an expression in any other expression. For instance, in an application we may have constant values, `if` or `let` expressions, or even lists in operator position, and `letrec`s, abstractions, variables or primitive operators in operand positions, or we may have primitive operators applied to incompatible operands. Typical examples of the latter kind are the expression $(+\ \text{"}abc\text{"}\ <>)$ which tries to add a character string to an empty list, or the expression `if` $(\text{"}abc\text{"}\ u)$ `then` 1 `else` 2 whose predicate expression cannot be evaluated to a Boolean value that is required to select between the consequent and the alternative.

That is to say, the syntax enables us to specify lots of expressions that are not very meaningful; nevertheless, they are correctly constructed and therefore perfectly legitimate. Unless we impose further restrictions on the construction of what we choose to consider legitimate expressions, we simply have to live with this situation and have to have the evaluation mechanism (or the semantics) take care properly of expressions that obviously make no sense.

3.2 The Evaluation of AL Expressions

We are now ready to define how AL expressions may be evaluated. All we need to do is to go through all the syntactical forms introduced in the preceding section and write down the rules for computing their values. As a first step, we will do this in a way that completely abstracts from the mechanisms of any underlying machinery to develop just some basic understanding as to how we have to proceed in principle.

For this purpose we introduce an abstract evaluator called EVAL. It may be considered a meta-function that maps every syntactical form into another one representing its value, or meaning. In this sense, EVAL may also be considered the semantic function of the language. It is defined by recursive application to either all or selected subexpressions of the syntactical forms, the selection being determined by the choice of a suitable evaluation strategy that can be expected to deliver result values with whatever may be considered a near-minimal computational effort.

We are going to define EVAL on the basis of a strategy that intuitively makes a lot of sense: it simply demands that generally the operands of applications be evaluated before the operators are applied to them, and that the cases in which this strategy causes problems be singled out and treated differently. This strategy is fairly straightforward to implement, yields a high runtime efficiency and is therefore taken as the best choice for the overwhelming majority of application problems. In fact, this operands-first strategy is implemented in almost all imperative programming languages, and in this context is also referred to as the call-by-value strategy.

Denoting the application of EVAL to some expression e by EVAL$[\![\ e\ ]\!]$ and going from top to bottom through the list of syntactical forms, we have

- for atomic expressions such as constants, variables and primitive operators,

  $$\text{EVAL}[\![\ atom\ ]\!] \ = \ atom \ ,$$

  i.e., they all are their own values (or meanings);
- for applications with unspecified expressions in operator and operand positions,

  $$\text{EVAL}[\![\ (e_0\ e_1\ \ldots\ e_n)\ ]\!] \ = \text{EVAL}[\![\ (\text{EVAL}[\![\ e_0\ ]\!]\ \text{EVAL}[\![\ e_1\ ]\!] \ldots \text{EVAL}[\![\ e_n\ ]\!])\ ]\!] \ ,$$

  i.e., EVAL is recursively driven in front of the subexpressions whose values need to be computed before the operator may be applied to the operands (which is effected by the outermost EVAL);[3]
- for `if_then_else` expressions (or conditionals)

  $$\text{EVAL}[\![\ \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ]\!] = \begin{cases} \text{EVAL}[\![\ e_1\ ]\!] \text{ if } \text{EVAL}[\![\ e_0\ ]\!] = \texttt{true} \\ \text{EVAL}[\![\ e_2\ ]\!] \text{ if } \text{EVAL}[\![\ e_0\ ]\!] = \texttt{false} \\ \texttt{if}\ \text{EVAL}[\![\ e_0\ ]\!]\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \quad \text{otherwise} \end{cases} ,$$

  that defines as the value of the entire expression either the value of e_1 or that of e_2 if e_0 evaluates to a Boolean value, but leaves the syntactical form intact and e_1 and e_2 unevaluated if the value of e_0 is something else;[4]
- for `let` expressions,

  $$\text{EVAL}[\![\ \texttt{let}\ v_1 = e_1\ \ldots\ v_n = e_n\ \texttt{in}\ e_0\ ]\!] = \text{EVAL}\ [\![\ e_0[\ v_1 \leftarrow \text{EVAL}\ [\![\ e_1\ ]\!]\ \ldots\ v_n \leftarrow \text{EVAL}\ [\![\ e_n\ ]\!]\ ]\ ]\!] \ ,$$

  i.e., it evaluates to the value of e_0 in which all occurrences of the `let`-bound variables have been substituted by the values of the respective defining expressions, with $v_i \leftarrow \text{EVAL}[\![\ e_i\ ]\!] \mid i \in \{1, \ldots, n\}$ denoting such substitutions;
- for anonymous abstractions occurring in other than operator positions of applications,

  $$\text{EVAL}[\![\ \texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0\ ]\!] \ = \ \texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ \text{EVAL}[\![\ e_0\ ]\!] \ ,$$

[3] An alternative evaluation strategy, on which we will focus extensively in later chapters, leaves the operand expressions of applications unevaluated, at least as long as it is not known what the value of the operator is going to be.

[4] It should be noted that the `if_then_else` construct is just a special syntactical form of an application $(e_0\ e_1\ e_2)$ that allows us to define EVAL so that just one of the operands e_1, e_2 is evaluated after the value of the operator e_0 has been determined, i.e., superfluous computations of things that are thrown away are avoided.

meaning that abstractions need to have their body expressions evaluated to determine their values, which must be done with care though;

- for lists,

$$\text{EVAL}[\![<>]\!] = <> \quad \text{and}$$
$$\text{EVAL}[\![< e_1 \ \ldots \ e_n >]\!] = <\text{EVAL}[\![e_1]\!] \ \ldots \ \text{EVAL}[\![e_n]\!] > \ ,$$

i.e., their values are recursively computed from the values of their subexpression while preserving their structure;

- for `letrec` expressions,

$$\text{EVAL}[\![\texttt{letrec} \ \ldots \ f_i = e_i \ \ldots \ \texttt{in} \ e_0]\!]$$
$$= \text{EVAL}[\![e_0[\ldots f_i \leftarrow \text{EVAL}[\![e_i[\ \ldots \ f_i \leftarrow \texttt{letrec} \ \ldots \ \texttt{in} \ f_i \ \ldots]]\!] \ldots]]\!] \ .$$

This rather complicated-looking definition may be read as follows: the value of the entire `letrec` is the value of its goal term e_0, computed by expanding all occurrences of the function identifiers f_i with (the values of) the right-hand sides e_i of their defining equations. These values must, in turn, be computed by substituting occurrences of the identifiers f_i in these expressions again, this time by copies of the full `letrec` expressions, which, however, have their goal expressions replaced by just the identifiers f_i themselves. These specialized syntactical forms $\texttt{letrec} \ldots f_i = e_i \ldots \texttt{in} \ f_i$ in fact represent the expressions e_i in unevaluated form, as the goal expressions f_i simply select them from the set of defining equations whenever evaluation of the entire `letrec` expression is enforced by driving the metafunction EVAL in front of them.
That is to say, copies of the complete `letrec` expression, specialized by the abstractions that need to be selected, are passed along as long as identifiers f_i are left in the actual goal expression. As soon as all of them have disappeared, the `letrec` expression is dropped as well since the function definitions are no longer needed for the computation to continue.

We have now gone through all the syntactical forms, but the definition of EVAL is far from being complete. So far, we have only covered the general case of applications that have unspecified expressions in operator and operand positions. What is still missing is the special cases where the expressions in operator position are (or evaluate to) abstractions or primitive operators. These applications define the real actions of transforming expressions into others.

Applications of n-ary abstractions to n operands expressions, i.e., the standard cases of full applications, are transformed by EVAL as

$$\text{EVAL}[\![(\texttt{lambda} \ v_1 \ \ldots \ v_n \ \texttt{in} \ e_0 \ \ e_1 \ \ldots \ e_n)]\!]$$
$$= \text{EVAL}[\![e_0[\ v_1 \leftarrow \text{EVAL}[\![e_1]\!] \ \ldots \ v_n \leftarrow \text{EVAL}[\![e_n]\!] \ ]]\!] \ .$$

They evaluate to the values of the abstraction bodies e_0 in which all occurrences of `lambda`-bound variables (the formal parameters of the abstractions) are substituted from left to right by the values of the operand expressions in the order in which they occur in the applications.[5]

However, the syntax of the language also allows for inequalities between the arity n of the abstraction and the number m of arguments. We could in such cases opt for the coward's approach of simply defining the value of such an application to be a character string signifying a mismatch:

- $\text{EVAL}[\![\ (\texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0\ \ e_1\ \ldots\ e_m)\ ]\!] \mid m \neq n\ =$
  "function of arity n receiving m arguments" .

The computation could then be aborted and this string be returned as the value of the entire expression.

Alternatively, we could try to do a little better and go for the following solutions:

- if the number m of arguments exceeds the arity n of the abstraction, we may define the value to be a new application whose operator is an expression resulting from the application of the n-ary abstraction to n arguments and whose operands are the remaining $m - n$ evaluated arguments:

$$\begin{aligned}&\text{EVAL}[\![\ (\texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0\ \ e_1\ \ldots\ e_m)\ ]\!] \mid m > n\ =\\ &\quad \text{EVAL}[\![\ (\text{EVAL}[\![\ e_0[\ v_1 \leftarrow \text{EVAL}[\![\ e_1\ ]\!]\ \ldots\ v_n \leftarrow \text{EVAL}[\![\ e_n\ ]\!]\ ]\ ]\!]\\ &\qquad\qquad \text{EVAL}[\![\ e_{n+1}\ ]\!]\ \ldots\ \text{EVAL}[\![\ e_m\ ]\!])\ ]\!]\ ;\end{aligned}$$

- if the arity n of the abstraction exceeds the number m of arguments, in which case we have a partial application, its value may be defined as

$$\begin{aligned}&\text{EVAL}[\![\ (\texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0\ \ e_1\ \ldots\ e_m)\ ]\!] \mid m < n\ =\\ &\quad \texttt{lambda}\ v_{m+1}\ \ldots\ v_n\ \texttt{in}\\ &\qquad\qquad \text{EVAL}[\![\ e_0[\ v_1 \leftarrow \text{EVAL}[\![\ e_1\ ]\!]\ \ldots\ v_m \leftarrow \text{EVAL}[\![\ e_m\ ]\!]\ ]\ ]\!]\ ,\end{aligned}$$

  i.e., it is an abstraction of arity $n - m$, in the body of which free occurrences of the bound variables $v_1, \ldots, v_n$ are now substituted by the evaluated arguments.

However, the case $m < n$ must be treated with particular care. As we have learned in Sect. 2.1, it could potentially cause naming conflicts between the remaining `lambda`-bound variables of the resulting abstraction and free-floating variables in the argument expressions that are being substituted under the `lambda` abstractor. The cases $n = m$ and $m > n$ and, as indicated earlier, also the evaluation of isolated abstractions are not entirely free of this

[5] Here again, we could alternatively leave the operand expressions unevaluated as it may very well happen that some of them are just thrown away without further action when the abstraction body is subsequently evaluated.

problem either, as abstraction body expressions may recursively contain other abstractions that may be penetrated by substitutions.

At this point in the discussion, we are not yet ready to deal with this problem, so we have to postpone proper treatment of it until later, and for now we must just keep in mind that the substitutions defined above must be considered naive in the sense that they do not resolve naming conflicts.[6]

To play it safe, we can take the conservative approach of alternatively defining the value of a partial application to be of the form

$$\begin{aligned}\text{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ \ e_1 \ldots e_m)\ ]\!] \mid m < n \\ = [\ \text{EVAL}[\![\ e_m\ ]\!] \ldots \text{EVAL}[\![\ e_1\ ]\!]\ \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ ]\ ,\end{aligned}$$

that has the evaluated arguments and the abstraction wrapped up in a construction embraced by the brackets [ ] that is called a closure. It represents the value of a new abstraction of arity $n - m$ without actually computing it, thus avoiding substitutions under abstractors and the naming conflicts that may come with them. Closures may be treated just like ordinary abstractions, i.e., they may be passed along as operators or operands of other applications and pick up more arguments until they reach the status of full appplications, at which point they may be evaluated by actually doing the (postponed) substitutions in the abstraction body then exposed.

Similar considerations regarding arities, though not involving name clashes, apply to applications of primitive operators as well. For the time being we define EVAL only for applications in which the arities of the operators (usually one or two) match the number of operands.

Turning first to (binary) arithmetic operators, which we summarily denote by *arith_op*, and using *num*, num_1 and num_2 to denote numbers, we have

$$\text{EVAL}[\![\ (arith\_op\ e_1\ e_2)\ ]\!] = \begin{cases} num \ \text{ if } num_1 = \text{EVAL}[\![\ e_1\ ]\!] \wedge num_2 = \text{EVAL}[\![\ e_2\ ]\!] \\ (arith\_op\ \text{EVAL}[\![\ e_1\ ]\!]\ \text{EVAL}[\![\ e_2\ ]\!]) \ \text{ otherwise} \end{cases},$$

i.e., the value of the application is a number if both e_1 and e_2 evaluate to numbers, and this value is given by applying the operator to both operand values; in all other cases the application simply remains as it is, other than that both e_1 and e_2 are now replaced by their values.

Likewise, for applications of relational operators, denoted as *rel_op*, and using str_1 and str_2 to denote character strings, and *bool* to denote Boolean values, we obtain

[6] A simple way out of this naming problem would be to make all variable names unique throughout the entire algorithm, which, however, might become difficult with growing size and complexity of the algorithm, even more so if it is assembled from independently written parts.

$$\text{EVAL}[\![\ (\mathit{rel\_op}\ e_1\ e_2)\ ]\!] = \begin{cases} \mathit{bool} \ \text{ if } \mathit{str}_1 = \text{EVAL}[\![\ e_1\ ]\!] \wedge \mathit{str}_2 = \text{EVAL}[\![\ e_2\ ]\!] \\ \qquad\qquad \text{or} \\ \qquad\qquad \mathit{num}_1 = \text{EVAL}[\![\ e_1\ ]\!] \wedge \mathit{num}_2 = \text{EVAL}[\![\ e_2\ ]\!] \\ \\ (\mathit{rel\_op}\ \text{EVAL}[\![\ e_1\ ]\!]\ \text{EVAL}[\![\ e_2\ ]\!]) \ \text{ otherwise} \ , \end{cases}$$

i.e., the value of the application is a Boolean value if the operands are either both character strings (in which case the operator uses lexical ordering to decide about truth or falsity) or both numbers, otherwise the application remains intact.

To make the story complete, we also give the definition of EVAL for a few operators that apply to lists. Here we make use of the fact that the component expressions of lists need not be evaluated in order to render primitive list functions applicable, i.e., EVAL needs to drive the evaluation of the operand expressions only to the point where it can be decided that they are lists or something other than applications:

$$\text{EVAL}[\![\ (\texttt{empty}\ e)\ ]\!] = \begin{cases} \texttt{true} \text{ if } \text{EVAL}[\![\ e\ ]\!] = <> \\ \\ \texttt{false} \text{ if } \text{EVAL}[\![\ e\ ]\!] = < e_1 \ldots e_n > \\ \\ (\texttt{empty}\ \text{EVAL}[\![\ e\ ]\!]) \ \text{ otherwise} \ , \end{cases}$$

$$\text{EVAL}[\![\ (\texttt{first}\ e)\ ]\!] = \begin{cases} e_1 \ \text{ if } \text{EVAL}[\![\ e\ ]\!] = < e_1 \ldots e_n > \\ \\ (\texttt{first}\ \text{EVAL}[\![\ e\ ]\!]) \ \text{ otherwise} \ , \end{cases}$$

$$\text{EVAL}[\![\ (\texttt{rest}\ e)\ ]\!] = \begin{cases} < e_2 \ \ldots \ e_n > \\ \qquad \text{if } \text{EVAL}[\![\ e\ ]\!] = < e_1\ e_2 \ldots e_n > \\ \\ (\ \texttt{rest}\ \text{EVAL}[\![\ e\ ]\!]\ ) \ \text{ otherwise} \ , \end{cases}$$

$$\text{EVAL}[\![\ (\texttt{append}\ e_1\ e_2)\ ]\!] = \begin{cases} < e_{11} \ldots e_{1n}\ e_{21} \ldots e_{2m} > \\ \text{if } \text{EVAL}[\![\ e_1\ ]\!] = < e_{11} \ldots e_{1n} > \\ \qquad \text{and } \ \text{EVAL}[\![\ e_2\ ]\!] = < e_{21} \ldots e_{2m} > \\ \\ (\texttt{append}\ \text{EVAL}[\![\ e_1\ ]\!]\ \text{EVAL}[\![\ e_2\ ]\!]) \ \text{ otherwise} \ . \end{cases}$$

In all four cases, the applications return the expected values if the operands are lists, i.e., a Boolean value if the list is empty or not empty, the first element of a list, a list of elements minus its first element, and a concatenated list of elements, respectively. Otherwise, the applications are left intact other than that the operands are evaluated to whatever their values are.

It is important to note at this point that the definition of EVAL for the primitive operators in fact includes dynamic type checking. Applications of these functions are evaluated only if the arguments are of compatible types,

otherwise they are left intact, except that the arguments themselves are evaluated. This is in compliance with what we said at the end of Chap. 2 about the preferred treatment of type inconsistencies detected at execution time.

The meta-function (or abstract evaluator) EVAL in fact defines an operational semantics for AL. It tells us not only what an expression means (or what its value is), but also how a person or a machine may proceed to compute this value. A close look at the definition of EVAL for applications reveals that we have some degree of freedom in choosing the order in which individual computational steps may be performed.

We observe that EVAL, when applied to an application, is driven in front of its subexpressions but that the EVAL in front of the application does not yet disappear. This means that the subexpressions need to be evaluated before the value of the entire application can be computed. However, since the subexpressions are syntactically completely independent of each other, they may be evaluated in any chosen order, even simultaneously, and their values will always be the same.

The EVALs are recursively propagated from outermost to innermost until subexpressions are encountered that, by definition, are their own values and let the EVALs disappear. The same happens with EVALs in front of applications whose components are all (atomic) values. These applications may either be evaluated to something else if they apply legitimate operators to compatible operands, e.g., arithmetic operators to numbers, or else be left unchanged, i.e., they too are their own values.

Thus, the EVALs that are recursively driven from outermost to innermost into an application may be considered demands that force the evaluation of its subexpressions. These demands are obviously satisfied from innermost to outermost as the EVALs disappear again.

The example in Fig. 3.1, which shows how EVAL goes about evaluating the arithmetic expression $(* \ (- \ 3 \ (+ \ 2 \ 3)) \ (/ \ 3 \ 6))$, may help us to understand what is going on here. The EVAL that initially sits in front of the entire application, demanding its evaluation (topmost line), spreads out over both of the outermost operand expressions $(- \ 3 \ (+ \ 2 \ 3))$ and $(/ \ 3 \ 6)$ but leaves the operator $*$ alone since it is already a value (second line from the top). Next, EVAL simultaneously penetrates the first operand to force the evaluation of $(+ \ 2 \ 3)$ and evaluates the second operand to 2, which lets the EVAL in front of it disappear as the demand is now satisfied (third line from the top). All other EVALs can now be satisfied from the inside out.

3.3 Summary

In this chapter, we have introduced an expression-oriented type-free algorithmic language AL that will be used as a reference language throughout the remainder of this text.

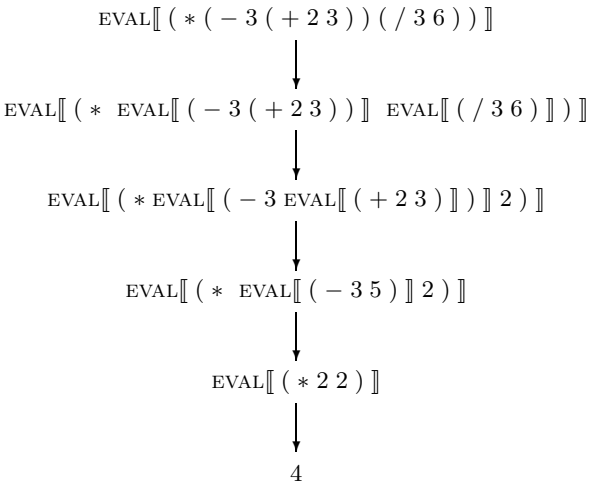


Fig. 3.1. The abstract evaluator at work

The syntax of AL provides several syntactical forms (or constructs) for expressions, the most important one being the application of an operator expression to one or more operand expressions, that may be freely combined with each other to design complex algorithms recursively from simpler parts. The semantics of this language is defined by an abstract evaluator (function) EVAL that prescribes how and in what (partial) order the value of an expression may be computed from the values of its component expressions.

Unfortunately, the syntactical freedom in designing AL algorithms invites a number of problems that have to be dealt with semantically but are not yet fully covered by the definition of EVAL.

Owing to the absence of a type system, there are many expressions that are perfectly legitimate syntactically but do not make much sense semantically. For instance, we may have applications whose operator expressions are (or evaluate to) to something other than legitimate operators, e.g., numbers or character strings, or are legitimate operators that are not type-compatible with the operands. EVAL considers such applications as constant expressions that are their own meanings, and leaves them as they are.

Syntactically, we may also have partial applications in which the arities of the operators exceed the number of operands. The precise definition of their evaluation must, for the time being, be left open for it may involve another problem: since variables may occur in all syntactical positions of expressions, specifically variables that are nowhere bound, there is always a potential for naming conflicts that render the results of computations dependent in intricate ways on the order of evaluating subexpressions.

Another open problem is the choice of the evaluation strategy, which primarily relates to the question of whether and to what extent operand expres-

sions ought to be evaluated before operators may be applied to them. EVAL demands that, generally, all operand expressions of applications be fully evaluated irrespective of the operators. This operands-first regime seems to make a lot of sense in many but not all cases. Unfortunately, in the case of recursive function calls it is bound to inflict runaway recursions even if termination conditions are correctly specified, say, in the form of selector expressions with multiple alternatives. The cause of the problem is that the operands-first strategy forces the evaluation of all alternatives, among them those that include the recursive calls, before the selection that throws away all but one of them can actually be performed. It takes a few exceptions to the operands-first rule in the definition of EVAL to deal with this and related issues of non-terminating or at least superfluous computations.

As an alternative, one could think of a strategy that generally applies operators to operands in unevaluated form and has the operators force the evaluation of operands whenever values are needed. It would eliminate the special cases from the definition of EVAL which in fact overrule the operands-first regime whenever there is a risk of doing superfluous work or of getting trapped in runaway recursions. However, there are also some strings attached to this operands-when-needed strategy: passing operands in unevaluated form to functions may inflict redundant computations insofar as the operands may have to be repeatedly evaluated in multiple places of substitution.

In order to fully understand the nature of and deal with all these problems, we have to undertake a little excursion into theory which we will do in the next chapter.

References

The language AL derives from a syntactically polished kernel of the type-free reduction language KIR described in [Kge94]. Its closest relatives in the world of function-based programming languages are LISP [McCA65], the more recent standard COMMONLISP [Ste84], its purified variant SCHEME [Dyb87], and to some extent also SML [MTHQ97]. The abstract evaluator EVAL, out of necessity, is similar to McCarthy's EVAL-APPLY interpreter as defined in [McCA65]. Other descriptions may also be found in [AS85, Kog91, FWH92].

4

The λ-Calculus

The λ-calculus, first published by Church in 1932, i.e., some time before the first computers came into being, is one of the mathematical models that were developed nearly at the same time in response to one of the problems raised by the mathematician Hilbert around the beginning of the 20th century. This problem concerns the question of whether there exists a general mechanical method of proving the truth or falsity of logical conjectures, which directly relates to the question of what is algorithmically computable. The other models include Schoenfinkel's and Curry's combinators (a special form of the λ-calculus), Goedel numbers, Post's production system, Kleene's recursive functions, Markov algorithms and, most prominently with regard to mechanical computations that can be peformed by digital machinery, the Turing machine. Though it is not quite clear what computability really means, some comfort can be derived from the fact that all of these models are in fact equivalent to each other. This has led to the well-known Church-Turing thesis, which states that intuitively or effectively computable problems are exactly those that are computable by Turing machines, or are Turing computable (and, for that matter, computable by the other models as well).

The λ-calculus is the model closest to algorithms and their evaluation as informally discussed in Chap. 2 and formalized in Chap. 3. In fact, it may be considered the paradigm of all programming languages as we know them today. It is primarily a theory of computable functions that deals with elementary properties of operators, with applications of operators to operands, and with the systematic construction of complex operators (or algorithms) from simpler components.

The core of the λ-calculus is based on little more than a well defined concept of variables, variable scoping and the orderly substitution of variables by expressions. The λ-calculus is a closed language, meaning that its semantics can be defined on the basis of the equivalence of expressions (or terms) of the calculus itself.

The term λ-calculus is derived from the notation that is used to represent functions. As the λ-calculus does not distinguish between operators and

operands other than by syntactical positions within an application, this notation greatly facilitates treating a function as an object of either kind. It also allows a precise definition of the notions of variable binding and of the binding status of variable occurrences in expressions.

4.1 λ-Calculus Notation

Functions of n variables $v_1, \ldots, v_n$ are in the λ-calculus denoted as

$$f \;=\; \lambda v_1 \ldots v_n.\; e_0 \quad .$$

Other than using the Greek letter λ instead of `lambda` and dropping the keyword `in`, this is exactly the same notation as introduced in Sect. 3.1 and earlier in Sect. 2.1. The symbol f on the left-hand side of this equation gives the function a name (or identifier) by which it may be referenced somewhere else,[1] and the expression on the right-hand side defines an abstraction of the variables $v_1, \ldots, v_n$ from the expression e_0, which is said to be the abstraction body.[2] The construct $\lambda v_1 \ldots v_n$ is an abstractor for free occurrences of the variables $v_1, \ldots, v_n$ in the function body e_0. A precise definition of what free variable occurrences are must be postponed until a little later. For now it may suffice to say that a variable u is free in the expression e_0 if it contains no abstractor for u preceding it.

An application of the function (or abstraction) named f to r argument expressions $e_1, \ldots, e_r$ has the form

$$(f\ e_1 \ldots e_r) \;=\; (\lambda v_1 \ldots v_n.e_0\ e_1 \ldots e_r) \quad ,$$

where r must not necessarily be equal to n.

The special case of applying an abstraction to the abstracted variables yields the abstraction body:

$$(\lambda v_1 \ldots v_n.e_0\ v_1 \ldots v_n) \;=\; e_0 \quad .$$

To keep the formal apparatus simple and concise, the λ-calculus considers, without loss of generality, only abstractions of one variable. This is due to a discovery by Schoenfinkel and Curry that renders it possible to represent n-ary abstractions as n-fold nestings of unary abstractions, i.e., we have

$$f \;=\; \lambda v_1 \ldots v_n.e_0 \;\equiv\; \lambda v_1.\lambda v_2 \ldots \lambda v_n.e_0 \quad .$$

Using this curried notation, the above application of f to r operands takes the form of an r-fold nesting of applications of unary abstractions:

[1] If the left-hand side is missing, then the abstraction is said to be nameless (or anonymous).

[2] The terms 'function' and 'abstraction' are in the following used synonymously.

$$(f\ e_1 \ldots e_r) \ = \ (\ldots((f\ e_1)\ e_2) \ldots e_r) \quad .$$

This reduces the construction of expressions (or terms) of the λ-calculus to just the following syntactical rules:

$$e \ =_s \ v \mid c \mid (e_0\ e_1) \mid \lambda v.e_0 \quad .$$

A λ-expression is either a variable, denoted as v, a constant value other than a variable, denoted as c, an application of a λ-expression e_0 to a λ-expression e_1, or an abstraction of a variable v from a λ-expression e_0, respectively. Expressions that emerge from a systematic application of these rules are also called the well-formed formulas of the λ-calculus; nothing else is a legitimate λ-expression.

An application $(e_0\ e_1)$ represents the result of applying e_0 to e_1; e_0 is said to be the expression in operator position and e_1 is said to be the expression in operand position of the application. Often e_0 and e_1 are also referred to as function and argument, respectively, which is somewhat misleading, insofar as the syntax allows any legitimate λ-expression to be in either syntactical position, but only abstractions and, for that matter, the primitives $+$, $-$, $*$, ... (which belong to the set of constant expressions) are truly functions. Nonetheless, if the expression in operator position is indeed a function, then we may legitimately call the expression in operand position its argument.

We are primarily interested in applications of the form $(\lambda v.e_0\ e_1)$ that have an abstraction in operator position and any legitimate λ-expression as its argument in operand position. These are the applications that tell us the full story about the role of variables, specifically of variable scoping and the substitution of variables, in the business of evaluating algorithmic expressions. In fact, it suffices to consider for this purpose only the pure λ-calculus whose set of constants is empty.[3] Without the primitive operators $+$, $-$, $*$, etc., abstractions are the only functions left in the game. Nevertheless, we have a complete formal model that provides the bare essentials necessary to investigate and reason about algorithmic computability. If so desired, we can even represent as λ-abstractions numbers, truth values or lists of such items as well as primitive value-transforming and structuring functions that operate on them. These abstractions admittedly look a little awkward, bearing no resemblance to their usual representation, particularly after they have undergone a few transformations, but they do the job in principle.

4.2 β-Reduction and α-Conversion

The beauty of the pure λ-calculus is that we need to be concerned with only one transformation rule since there are only applications of abstractions left to worry about. As we have learned in Sect. 3.2, this rule is intended to replace

[3] Otherwise, the λ-calculus is said to be applied.

such applications by the abstraction bodies in which all free occurrences of the λ-bound variables are substituted by the respective operand (or argument) expressions, which we denote by:

$$(\lambda v.e_0\ e_1)\ \rightarrow_\beta\ e_0[\, v \leftarrow e_1\,]\ .$$

The subscript attached to the arrow that points to the right gives the rule its name: it is called the β-reduction or β-contraction rule of the λ-calculus which is said to simplify, contract or reduce the β-redex $(\lambda v.e_0\ e_1)$ to its reductum or contractum $e_0[\, v \leftarrow e_1\,]$.

Unfortunately, this rule is not as simple as it may appear at first glance. As we have learned in Sect. 2.1, there are problems with regard to bound variables in abstractions and equally named free-floating variables in operand expressions, in which cases substitutions cannot be performed without some corrective actions to one of the variables. They concern the binding status of variables which must remain invariant against β-reductions in order to guarantee determinacy of results irrespective of the choice of variable names.

If the substitutions would be carried out naively, i.e., with the operand terms literally as they are, as in

$$(\lambda u.\lambda v.u\ w)\ \rightarrow\ \lambda v.w\ \text{ and }\ (\lambda u.\lambda v.u\ v)\ \rightarrow\ \lambda v.v\ ,$$

we would obtain as contractum in the first case the abstraction $\lambda v.w$ and in the second case the abstraction $\lambda v.v$. Obviously, the choice of the variable in the operand position would result in two entirely different functions. In the first case we would get a constant function that, irrespective of the operand to which it may be applied, returns the function body w since the operand can nowhere be substituted. However, in the second case we would get an identity function that always reproduces the operand expression:

$$(\lambda v.w\ a)\ \rightarrow\ w\ \text{ and }\ (\lambda v.v\ a)\ \rightarrow\ a\ .$$

The problem comes about in the second case where the free-floating variable v is naively substituted into the scope of the abstractor λv and thus becomes parasitically bound by it, whereas in the first case we substitute the variable w that remains unaffected by the abstractor λv and thus preserves its binding status as being free.

We could decide to accept these parasitic bindings, that in fact are caused by name clashes or naming conflicts as discussed in Sect. 3.2, as part of the game and possibly even use them as dirty little tricks to take advantage of some algorithmic shortcuts here and there. Unfortunately, they also destroy two very useful and important properties of the λ-calculus that, as we will see later, guarantee a very orderly behavior with regard to evaluation strategies and should therefore not be given up easily.

To prevent name clashes from occurring, we could play it safe and simply demand that all variables within a λ-expression be named differently. However, this does not appear to be a very realistic requirement for pragmatic

reasons. With increasing numbers of variables in complex algorithms it may become increasingly difficult to invent variable names that somehow relate to the intended purpose or to a convention, for example using $i,\ j,\ k$ for indexing, and keep them distinguishable. Making variable names unique by automated means, say by enumeration, may be a practical alternative provided that the new names somehow relate to the original ones, i.e., do not alienate the algorithm beyond recognition. This is particularly important if we intend to have a machine execute algorithms step by step, as described in Sect. 2.1, and wish to inspect intermediate expressions, say, for validation purposes. Here we would prefer to see the variables used in the original expression, not variables that may have been invented by the machine.

We will first have a look at the classical solution of the naming problem and see how far we can get with the idea of maintaining the original variable names. To do so, we need to set out with a precise definition of what is meant by free and bound variables (or by the binding status of variables), and by variable scoping.

With V denoting the set of variables, we may define the set of free variables FV of an expression e simply by going through the three syntactical forms of the pure λ-calculus and specifying what the free variables are in these cases. This gives[4]

$$FV(\ e\ ) = \begin{cases} \{\ v\ \} \ \text{ if } \ e =_s v \in V \\ FV(\ e_0\ ) \ \cup \ FV(\ e_1\ ) \ \text{ if } \ e =_s (e_0\ e_1) \\ FV(\ e_0\ ) \setminus \{\ v\ \} \ \text{ if } \ e =_s \lambda v.e_0 \ . \end{cases}$$

This recursive definition says that the set contains just the variable v if v is the entire expression e, that it is the union of the sets of free variables of the operator and operand if the expression is an application, and if the expression is an abstraction it is the set of free variables of the abstraction body without the variable that is bound in it.

A complementary definition can be given for the set of bound variables BV of an expression e as:

$$BV(\ e\ ) = \begin{cases} \emptyset \ \text{ if } \ e =_s v \in V \\ BV(\ e_0\ ) \ \cup \ BV(\ e_1\ ) \ \text{ if } \ e =_s (e_0\ e_1) \\ BV(\ e_0\ ) \ \cup \ \{\ v\ \} \ \text{ if } \ e \ =_s \ \lambda v.e_0 \ . \end{cases}$$

This set is empty if e is just a variable, it is the union of the bound-variable sets of its components if e is an application, and if the expression is an abstraction

[4] We use here again the symbol $=_s$ for syntactical equality (equality by construction) of two λ-expressions.

it is the set of variables bound inside the abstraction body plus the variable bound by the abstraction itself.

With these definitions at hand, we can now say that a variable v is free in an expression e if and only if $v \in FV(\ e\ )$, and that it is bound in e if and only if $v \in BV(\ e\ )$. In an abstraction $\lambda v.e$, we call the body e the scope of the abstractor λv, which means that all free occurrences of v in e are bound by λv.

An example may help to clarify this terminology:

$$\underbrace{\underbrace{(\lambda u.\,(\lambda v.\,(\lambda z.\,\underbrace{(z\ (v\ u))}_{\text{scope of } \lambda z}\ v\ )\ u\ )}_{\text{scope of } \lambda v}\ w\ )}_{\text{scope of } \lambda u}\ .$$

Here the variable w is free in the entire expression since there exists no abstractor for it. The variable u is bound in the abstraction $\lambda u.(\ \ldots\ )$ but free in its body, which happens to be the scope of the abstractor λu. Likewise, the variables v and z are bound in the abstractions $\lambda v.(\ \ldots\ )$ and $\lambda z.(\ \ldots\ )$, respectively, but free in their bodies (or in the scopes of λv and λz). Thus, a variable may be both free and bound in an expression, depending on the scope that is being considered.

An abstraction whose set of free variables is empty is said to be closed or a combinator; otherwise it is said to be open. Of particular importance with respect to the implementation of λ-calculus-based programming languages, as we will see later on, are so-called supercombinators. These are closed abstractions whose body expressions may recursively contain only closed abstractions (or supercombinators).

Now we are ready to define precisely how the substitution involved in a β-reduction

$$(\lambda v.e_b\ e_a)\ \rightarrow_\beta\ e_b[\ v \leftarrow e_a\ ]$$

must be performed: the right-hand side of this transformation rule must prescribe the substitution of all free occurrences of the variable v in the expression e_b by the expression e_a. This definition is given in Fig. 4.1.

The last three cases are the interesting ones. They prescribe what has to be done if the expression e_b in which substitutions of free occurrences of v are to take place is an abstraction. If this abstraction binds v, then it remains unchanged as there are no free occurrences of v in it. If it binds another variable u, then v can be naively substituted by e_a if there is no free occurrence of u in e_a that would become parasitically bound, or if we have the trivial case that v does not occur in e_b, i.e., nothing is substituted anywhere.

The complementary case where the abstraction binds u, with v occurring free in the abstraction body e_0 and with u occurring free in e_a, causes a naming conflict: when e_a would be substituted naively in e_0, free occurrences of u in e_a would become parasitically bound by the abstractor λu and thus change their binding status.

$$
e_b[\, v \leftarrow e_a \,] = \begin{cases}
e_a \quad \text{if} \quad e_b =_s v \in V \\
\\
u \quad \text{if} \quad e_b =_s u \in V \quad \text{and } v \neq_s u \\
\\
(e_0[\, v \leftarrow e_a \,]\; e_1[\, v \leftarrow e_a \,]) \quad \text{if} \quad e_b =_s (e_0\; e_1) \\
\\
\lambda v.e_0 \quad \text{if} \quad e_b =_s \lambda v.e_0 \\
\\
\lambda u.e_0[\, v \leftarrow e_a \,] \\
\qquad\qquad \text{if} \quad e_b =_s \lambda u.e_0 \\
\qquad\qquad \text{and} \quad u \notin FV(\, e_a \,) \text{ or } \; v \notin FV(\, e_b \,) \\
\\
\lambda w.e_0[\, u \leftarrow w \,][\, v \leftarrow e_a \,] \\
\qquad\qquad \text{if} \quad e_b =_s \lambda u.e_0 \\
\qquad\qquad \text{and} \quad u \in FV(\, e_a \,) \text{ and } v \in FV(\, e_b \,) \\
\qquad\qquad \text{and } w \in V \quad \text{and} \quad w \notin FV(\, e_a \,) \cup FV(\, e_b \,)
\end{cases}
$$

Fig. 4.1. Classical definition of the β-reduction rule

There are two ways out of this dilemma. We could either change the free variable v in e_a, say to w, or change the variable bound by the abstraction from u to w to take it out of the conflict. In either case, w should be a fresh variable that is used nowhere else in the original β-redex. The traditional solution is the latter, which is included as the last case in the above definition. It is the more convenient one as the renaming concerns only variables occurring within the scope of the application.[5]

The renaming transformation

$$\lambda u.e_0 \;\rightarrow_\alpha\; \lambda w.e_0[\, u \leftarrow w \,]$$

that is used here is called an α-conversion. It may be realized by application of an α-conversion function to the abstraction whose bound variable needs to be changed:

$$(\lambda v.\lambda w.(v\; w)\; \lambda u.e_0\,) \quad .$$

This transformation proceeds, by application of the above rules, in two steps

$$\text{first to} \;\; \lambda w.(\lambda u.e_0 \;\; w) \quad \text{and then to} \quad \lambda w.e_0[\, u \leftarrow \; w] \quad .$$

To illustrate the application of the substitution rules for $e_b[\; v \leftarrow e_a \;]$, Fig. 4.2 shows the stepwise reduction of the above example in slightly modified form: we replace the free variable w by z to create a name clash when the β-redices are systematically reduced from outermost to innermost. In this

[5] If we were to rename the free variable then we would have to deal with the problem that it could be bound somewhere in a larger context surrounding the application, i.e., renaming would concern all occurrences of the variable within the scope of this abstractor.

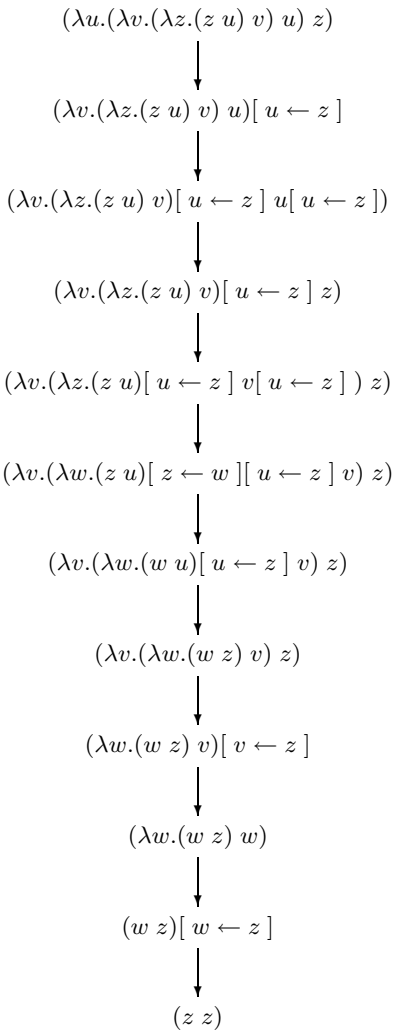


Fig. 4.2. A sequence of β-reductions

sequence, the last of the substitution rules is applied to the fifth expression from the top to rename to w the bound variable z of the abstraction $\lambda z.(z\ u)$ in order to get around a name clash with the free variable z that must be substituted for u. But this is just what we did not wish to do since now a variable has entered the game that was not known in the original expression but has been brought in from nowhere (e.g., by the system that performs the computation), thus changing the appearance (but not the meaning) of the abstraction $\lambda z.(z\ u)$ to $\lambda w.(w\ u)$. However, in this particular case we are lucky: if we just look at the original and final expression, and ignore the

steps in between, the renaming goes unnoticed since the normal form is an application of the original variable z to itself that preserves its binding status of being free throughout the entire sequence of reduction steps.

The story looks a little different for λ-expressions that reduce to abstractions, as for instance

$$(\lambda u.(\lambda v.\lambda z.((z\ u)\ v)\ z)\ v)\ .$$

When performing β-reductions, say from outermost to innermost, we encounter two naming conflicts, the first when substituting the outer argument v under λv, and the second when substituting the inner argument z under λz. If we rename v to x first and then z to y, we get the abstraction $\lambda y.((y\ v)\ z)$ as result. However, if we were to reverse the order in which we use x and y, we would get the abstraction $\lambda x.((x\ v)\ z)$, which is the same as before, except that the name for the bound variable has changed.

Though the choice of names for bound variable is completely irrelevant, doing several β-reductions that require renaming in a larger context may nevertheless be quite confusing as it may alienate, with regard to the variables originally used, the resulting expression beyond recognition.

To avoid these annoying naming problems, we have to come up with a smarter idea for representing the binding status of variables and for performing β-reductions that under all circumstances preserves the variable names as introduced in the initial expression.

4.3 An Indexing Scheme for Bound Variables **

When specifying an abstraction, it does not really matter what names for bound variables we choose. They merely relate abstractors to syntactical positions in abstraction bodies, or are placeholders, where argument expressions need to be substituted. We might say that this relationship defines a binding structure.

In this respect, the following two abstractions are syntactically equivalent modulo α-conversion of variable names:

$$\lambda u.\lambda v.\lambda w.(((w\ v)\ u)\ (x\ u)) \ =_s\ \lambda x_1.\lambda x_2.\lambda x_3.(((x_3\ x_2)\ x_1)\ (x\ x_1))\ .$$

When we apply the two abstractions to the same arguments, we obtain in both cases exactly the same result since two expressions that are syntactically equivalent are also semantically equivalent.

The position of an abstractor in a sequence of abstractors also identifies, in a nesting of applications, the nesting level from which the argument is to be taken. We will call this the substitution structure. The resulting structure, which associates abstractors with occurrences of variables in the abstraction body on the one hand and with operand positions in a nesting of applications on the other hand, may be depicted graphically as in Fig. 4.3.

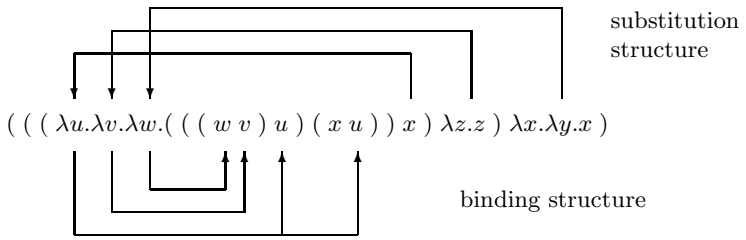


Fig. 4.3. Substitution and binding structures of an abstraction application

We also give in Fig. 4.4 the sequence of β-reductions that evaluates these nested applications, because we will need it for comparison later on. This sequence performs, from innermost to outermost, the β-reductions that substitute in the abstraction body x for u, $\lambda z.z$ for v, and $\lambda x.\lambda y.x$ for w, and then proceeds to evaluate the instantiated body, returning as the result the application $(x\ x)$ of the free variable x to itself.

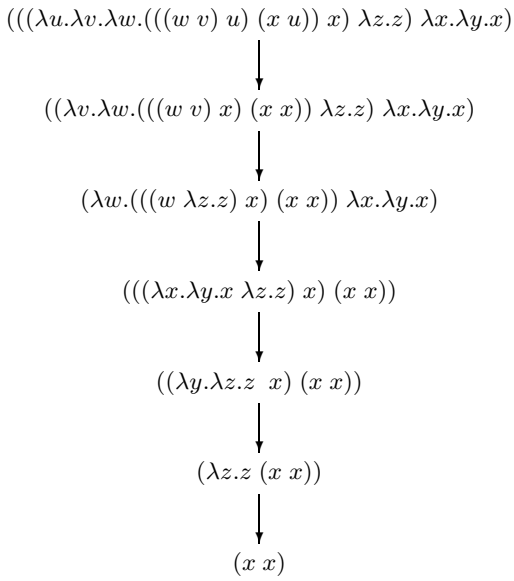


Fig. 4.4. Reduction sequence for the application in Fig. 4.3

We will now take the renaming of variables one step beyond naming them all x and distinguishing them by subscripts. We give them all the same names, say z, and define the binding structure by so-called unbinding operators (or protection keys) that we put in front of the variable occurrences in the abstrac-

tion body. These unbinding operators are complementary to the abstractors in that, loosely speaking, they undo bindings: if a variable occurrence z is preceded by n of these operators, denoted as $\underbrace{//\ldots/}_{n}\ z$, then it is protected against the innermost n abstractors λz or, to put it another way, there are n abstractors λz between the variable occurrence and the abstractor λz that actually binds it.

Using this notation, the above abstraction may be equivalently represented as

$$\lambda u.\lambda v.\lambda w.(((w\ v)\ u)\ (x\ u)) =_s \lambda z.\lambda z.\lambda z.(((z\ /z)\ //z)\ (x\ //z))\ .$$

All bound variables are now named z, and occurrences of z replacing the original variable u in the abstraction body are now preceded by two keys $//$ to protect them against the innermost two abstractors λz and hook them up to the outermost abstractor. Likewise, what was originally the variable v is now z preceded by one key to protect it against the innermost abstractor, whereas the single occurrence of z that replaces what was originally w carries no protection key since it needs to be bound by the innermost λz. The x remains as it is since it is free in the entire abstraction.

The trouble with these protection keys is that they need to be dynamically changed as β-reductions are performed, to preserve the original binding structures by the remaining abstractors and variable occurrences. If abstractors disappear in between, then protection keys have to be removed, and if free variables penetrate the scopes of abstractors for the same variable name, then the number of protection keys has to be stepped up accordingly.

We will see how this works by applying our abstraction to the three operand expressions that have their variables changed to z too.

To make things a little more interesting, we add two more abstractors in front of the entire application, of which the outermost binds the free variable x. When these variables are changed to z as well, the z that now takes the place of the x receives four keys to protect it against the three abstractors $\lambda z.\lambda z.\lambda z$ and against the inner of the two added abstractors:

$$\lambda z.\lambda z.(((\lambda z.\lambda z.\lambda z.(((z\ /z)\ //z)\ (///z\ //z))\ /z)\ \lambda z.z)\ \lambda z.\lambda z./z)\ .$$

For the same reason, the z in the innermost operand position that replaces the x receives one key to relate it to the very same abstractor to which the $////z$ in the abstraction body is hooked up.

The sequence of β-reductions shown in Fig. 4.5 that evaluates this application performs exactly the same number of steps as the sequence in Fig. 4.4, producing the same intermediate expressions just in a different representation.

The first of these reduction steps contains in a nutshell everything we need to know about the manipulation of the protection keys when performing β-reductions. The β-redex that is under consideration here applies the abstraction $\lambda z.\lambda z.\lambda z.(\ \ldots\ )$ to the argument $/z$.[6] In doing so, it drops the

[6] The redices to be contracted next are underlined in this figure.

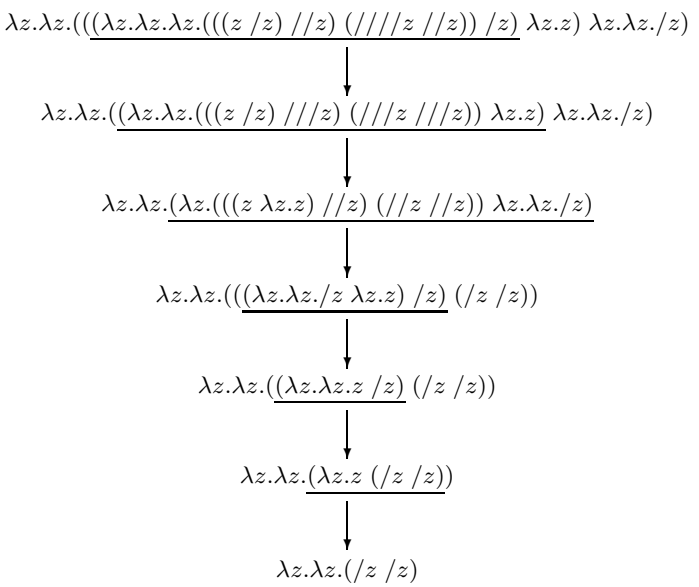


Fig. 4.5. The same reduction sequence as in Fig. 4.4, with all bound variables named z, and with binding distances maintained by means of protection keys

first λz and substitutes $/z$ for all occurrences of variables bound to it in the abstraction body, which are the two occurrences of $//z$. Now, substituting the $/z$ under the two remaining abstractors means that two more protection keys must be added to it, i.e., the two occurrences of $//z$ must be replaced by $///z$ to keep the correct distance relative to the outermost λz, which is in front of the entire application. Moreover, the variable occurrence $////z$ that is inside the abstraction but is bound by the outermost abstractor λz must drop one protection key since one of the abstractors in between has gone. This leaves three occurrences of $///z$ under the remaining abstraction that are bound by the outermost λz.

The protection keys of these variables are decremented to one by two more β-reductions, and the result is a self-application of $/z$ that has maintained its status of being bound to the outermost of the two abstractors that have been added in front and have never participated in β-reductions.

We can now be a little more specific about the manipulation of these protection keys by first defining the binding status of a variable that is preceded by them.

Let $\underbrace{/\ldots//}_{i} v =_s /^{(i)}v \mid i \in \{0,\ 1, \ldots\}$ be a variable occurrence with i-fold protection inside a λ-expression e, and let $j \in N_0$ enumerate, from the variable occurrence outwards and beginning with $j = 0$, the abstractors λv

surrounding it, then, relative to the j-th abstractor, this variable occurrence is said to be

$$\rightarrow \textsf{free if } i > j \ , \qquad \rightarrow \textsf{bound if } i = j \ , \qquad \rightarrow \textsf{shadowed if } i < j \ .$$

The protection index i of a variable occurrence denotes what may be referred to as its binding index or as its binding distance (to the binding abstractor).

With these binding attributes at hand, we can give an initial informal definition of the β-reduction rule that specifies what needs to be done in principle.

Given a redex $(\lambda v.e_b \ e_a)$, β-reduction returns an expression e'_b that is obtained from e_b in that an occurrence of $/^{(i)}v$

- in the abstraction body e_b
  - decrements the number i of protection keys by one if free,
  - is substituted by e_a if bound,
  - remains unchanged if shadowed

  relative to the abstractor λv that is in front of e_b;
- in the operand e_a
  - increments the number i of protection keys by some value k if free or bound relative to the innermost λv that surrounds the application $(\lambda v.e_b \ e_a)$, and if it penetrates the scopes of k nested abstractors λv when substituted in e_b,
  - remains unchanged if shadowed.

Transforming a λ-expression with freely choosen variables into an equivalent expression that has all bound variables named the same, say z as in our example, can be accomplished by the application of the α-conversion function $\lambda v.\lambda z.(\ v\ z\ )$ to all abstractions of one variable. In the case of our example, these applications would have to be inserted into the abstraction thus (the items that are added are underlined):

$$\underline{(\lambda v.\lambda z.(v\ z)}\ \lambda u.\underline{(\lambda v.\lambda z.(v\ z)}\ \lambda v.\underline{(\lambda v.\lambda z.(v\ z)}\lambda w.(((w\ v)\ u)\ (x\ u)))))\underline{)} \ .$$

Performing these α-conversions from innermost to outermost and using the β-reduction rule as just defined produces the sequence of reduction steps shown in Fig. 4.6. This sequence terminates with an abstraction that has all occurrences of bound variables turned into z and the right number of protection keys attached to them. The variable x is left unchanged since there is no abstractor for it in the entire expression.

4.4 The Nameless Λ-Calculus

Once we have given all bound variables the same name and distinguished different bindings just by the number of protection keys that precede them, we might as well drop variable names altogether and work with nameless

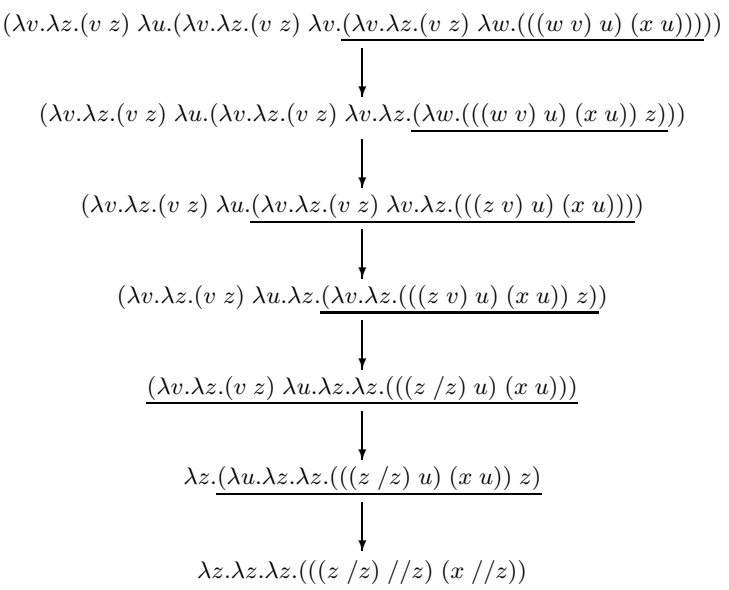


Fig. 4.6. Systematic α-conversion of λ-bound variables of the abstraction of Fig. 4.4 (the redices that are actually contracted are underlined)

abstractors Λ in combination with binding indices $\#i$ instead. Our example abstraction with which the sequence of α-conversions in Fig. 4.6 terminates then assumes the equivalent form

$$\Lambda.\Lambda.\Lambda.(((\#0\ \#1)\ \#2)\ (x\ \#2)) \quad ,$$

and β-reduction follows the same rules as defined above, except that it manipulates these indices rather than numbers of protection keys.

We will refer to this variant of the λ-calculus, which was invented by Berkling and deBruijn independently, as the Λ-calculus of nameless dummies, or simply the nameless Λ-calculus. It constitutes a considerable departure, with regard to representation, from the λ-calculus that uses variables to define binding structures. Λ-expressions may be less readable to human beings (at least, it takes some getting used to them) but they are decidedly more suitable for interpretation by machines: β-reductions replace fairly complicated name-handling operations by simple index manipulations, and – as we will see later on – the indices relate more or less directly to accesses to a runtime environment.

The syntax of the pure Λ-calculus is essentially the same as that of the λ-calculus, except that bound variables are replaced by binding indices and abstractor symbols λv are replaced by Λ:

$$e =_s \#i \mid (e_0\ e_1) \mid \Lambda.e_0 \quad \text{with} \quad i \in \{0,\ 1, \ldots\} \quad .$$

On the basis of these syntactical figures, we can now precisely define the β-reduction rule with nameless dummies in an operational form that may serve as a recipe for its implementation. Following the above informal definition, we obviously need operations that increment, decrement and substitute binding indices $\#i$ in some Λ-expression e relative to abstractors that are away from the indices by distances of k intervening abstractors.

The operator $p^{(k)}$ increments all free occurrences of binding indices in e by one:

$$p^{(k)}e = \begin{cases} \#i & \text{if } e =_s \#i \text{ and } k > i \\ \#(i+1) & \text{if } e =_s \#i \text{ and } k \leq i \\ (p^{(k)}e_0 \;\; p^{(k)}e_1) & \text{if } e =_s (e_0\ e_1) \\ \Lambda.p^{(k+1)}e_0 & \text{if } e =_s \Lambda.e_0 \;\; . \end{cases}$$

The dual operator $q^{(k)}$ decrements all free occurrences of binding indices in e by one:

$$q^{(k)}e = \begin{cases} \#i & \text{if } e =_s \#i \text{ and } k \geq i \\ \#(i-1) & \text{if } e =_s \#i \text{ and } k < i \\ (q^{(k)}e_0 \;\; q^{(k)}e_1) & \text{if } e =_s (e_0\ e_1) \\ \Lambda.\ q^{(k+1)}e_0 & \text{if } e =_s \Lambda.e_0 \;\; . \end{cases}$$

Finally, the operator $s^{(k)}$ substitutes in an expression e_0 occurrences of binding indices that are bound at a binding distance k by the expression e_1:

$$s^{(k)}e_1\ e_0 = \begin{cases} \#i & \text{if } e_0 =_s \#i \text{ and } i \neq k \\ e_1 & \text{if } e_0 =_s \#i \text{ and } i = k \\ (s^{(k)}e_1\ e_a \;\; s^{(k)}e_1\ e_b) & \text{if } e_0 =_s (e_a\ e_b) \\ \Lambda.s^{(k+1)}p^{(0)}e_1\ e_a & \text{if } e_0 =_s \Lambda.e_a \;\; . \end{cases}$$

Using these operators, β-reduction can now be defined as

$$(\Lambda.e_0\ e_1) = q^{(0)}(s^{(0)}(p^{(0)}e_1)\ e_0) \quad .$$

This may look a little strange at first sight, but if we apply the operators one by one from innermost to outermost, we can see what happens.

To begin with, we have to realize that on the right-hand side of this equation we have the abstraction $\Lambda.e_0$ stripped from its abstractor without modifying e_0 itself, i.e., all binding indices remain unchanged as if the abstractor was still in effect.

Applying $p^{(0)}$ to the argument term e_1 now simply increments free occurrences of binding indices in it by one to prepare it for the application of $s^{(0)}$. This application in fact moves $p^{(0)}e_1$ across the abstractor that preceded e_0, assuming it was still there, and substitutes it for all occurrences of binding indices $\#0$ in e_0. The operator $q^{(0)}$ finally does the equivalent of removing the abstractor Λ in that it decrements all free occurrences of binding indices in the resulting term by one.

At this point it may be helpful to make it a little more plausible that the indices are manipulated correctly. It is easy to see that in $(p^{(0)}e_1)$ all binding indices must have values greater than zero. The same must be true for the term $(s^{(0)}\ (p^{(0)}e_1)\ e_0)$ since after substitution of $(p^{(0)}e_1)$ for free occurrences of indices $\#0$ there are no zero indices left in e_0 either. Thus, when $q^{(0)}$ is applied, all indices in the resulting expression are guaranteed to have non-negative values.

We may convince ourselves that this works as expected by means of the β-reduction

$$(\Lambda.\Lambda.(\#0\ \#1)\ \#2)\ \rightarrow_\beta\ \Lambda.(\#0\ \#3)\ ,$$

which is depicted in Fig. 4.7 as a sequence of transformation steps effected by application of the operators $p^{(0)}, q^{(0)}$ and $s^{(0)}$.

The essence of performing β-reductions by means of these operators is that they are based solely on simple index increments and decrements that may be distributed recursively over the expressions involved and applied locally as the evaluation proceeds, and that the indices in a natural way become the focus of activity.

To highlight another interesting property of the Λ-calculus, we now return to the abstraction used as an example earlier and apply it to operands that are or include binding indices themselves:

$$(((\Lambda.\Lambda.\Lambda.(((\#0\ \#1)\ \#2)\ (x\ \#2))\ (\#2\ \#3))\ \#2)\ \#1)\ .$$

The indices in the operand expressions are assumed to be bound somewhere outside, in some context surrounding this application that is not of interest.

Looking at the sequence of reductions of this application in Fig. 4.8, we note that the indices that make up the arguments are stepped up by the number of Λ-abstractors that they cross when they are substituted in the abstraction body, but that these indices are decremented again as these abstractors disappear due to subsequent β-reductions. Once all β-reductions have been done, the indices are again the original ones in their places of substitution, i.e., they have in fact not changed at all.

The free variable x in the abstraction body behaves like a constant value that remains unaffected by all β-reductions.

We note that the body of the abstraction $\Lambda.\Lambda.\Lambda.(((\#0\ \#1)\ \#2)\ (x\ \#2))$ contains only indices smaller than the number of abstractors preceding it, i.e., the entire abstraction is an expression whose set of free indices is empty, or it contains only bound and shadowed indices according to our definition. We

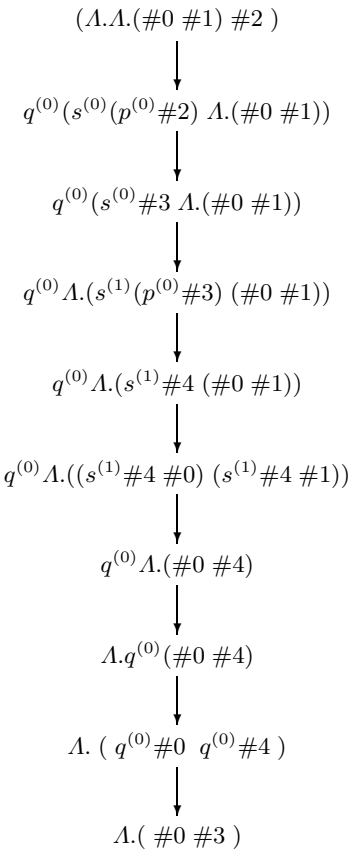


Fig. 4.7. β-reducing a Λ-expression by means of the operators $p^{(0)}, q^{(0)}$ and $s^{(0)}$

refer to such an expression as being closed or, in this particular case, also as a supercombinator since the abstraction body trivially contains no further abstractions either.

As the example shows, the nice part about such closed abstractions is that applications to full sets of operands can obviously be reduced without worrying about binding indices that may have to be changed in argument expressions when all substitutions (β-reductions) are performed in one conceptual step.

This is a very important property which we will exploit to advantage later on in the design of abstract λ-calculus machines. It renders time-consuming full-fledged β-reductions at the machine level largely superfluous as almost all of them can in fact be performed naively, i.e., by substituting operands as they are. Moreover, the indices lend themselves directly to the structure of and accesses to a runtime environment that holds instantiations of Λ-bound variables.

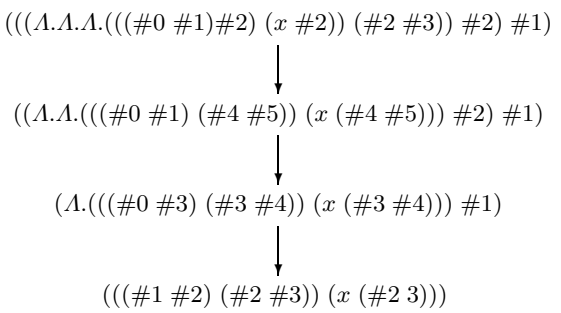


Fig. 4.8. Reducing the application of a closed abstraction

4.5 Reduction Sequences

It is now time to learn more about the properties of sequences of β-reductions.[7]

Given two λ-expressions e and e', we can say that e is β-reducible to e', denoted as $e \longmapsto_\beta e'$, if and only if e can be transformed into e' by a finite, possibly empty sequence of β-reductions and α-conversions. It produces a sequence $e_0, \ldots, e_n$ of λ-expressions with $e_0 =_s e$ and $e_n =_s e'$ such that for all indices $i \in \{0, \ldots, n-1\}$ we have either $e_i \rightarrow_\beta e_{i+1}$ or $e_i \rightarrow_\alpha e_{i+1}$.

On the basis of such sequences, we may also define that two λ-expressions e and e' are semantically equivalent, denoted as $e = e'$, if and only if e can be transformed into e' by a finite, possibly empty sequence of β-reductions, reverse β-reductions, and α-conversions, i.e., for all indices $i \in \{0, \ldots, n-1\}$ we must have $e_i \rightarrow_\beta e_{i+1}$ or $e_{i+1} \rightarrow_\beta e_i$ or $e_i \rightarrow_\alpha e_{i+1}$.

The objective of reducing a λ-expression e is to transform it to some expression e^{NF} that contains no more redices, or to which no more β-reduction rules are applicable. This expression is called the normal form (or the value) of the expression e. If it takes a finite sequence of β-reductions to get from e to e^{NF}, then e^{NF} is, of course, also the normal form of all intermediate λ-expressions.

A complex λ-expression that contains several redices generally allows to choose among several alternative sequences of β-reductions. The trouble with these choices is that, starting from some initial expression, we may reach a normal form

- with all possible sequences eventually, i.e., after finitely many β-reductions have been performed in any order, if we are lucky;
- with none of the possible sequences, because a normal form does not exist, i.e., none of the sequences terminates after finitely many steps;
- with some of the possible sequences but not with others, i.e., we have some sequences that terminate after finitely many steps, but others that do not.

[7] To render the expressions more readable, we return here to the λ-calculus of variables.

The latter case is the interesting one because it calls for a strategy that makes sure that β-reductions are performed in an order that terminates with a normal form, if it exists at all.

An example of the first kind is the expression $(\lambda u.(\lambda w.(\lambda w.u\ u)\ u)\ w)$ that allows three alternative sequences, of which two – incidently – require the resolution of naming conflicts by means of protection keys:

- doing the redices from outermost to innermost, we obtain:

$$(\lambda u.(\lambda w.(\lambda w.u\ u)\ u)\ w) \rightarrow_\beta (\lambda w.(\lambda w.//w\ /w)\ w) \rightarrow_\beta (\lambda w./w\ w) \rightarrow_\beta w\ ;$$

- starting with the redex in the middle, we obtain:

$$(\lambda u.(\lambda w.(\lambda w.\ u\ u)\ u)\ w) \rightarrow_\beta (\lambda u.(\lambda w.u\ u)\ w) \rightarrow_\beta (\lambda w./w\ w) \rightarrow_\beta w\ ;$$

- doing the redices from innermost to outermost, we get:

$$(\lambda u.(\lambda w.(\lambda w.u\ u)\ u)\ w) \rightarrow_\beta (\lambda u.(\lambda w.u\ u)\ w) \rightarrow_\beta (\lambda u.u\ w) \rightarrow_\beta w\ ;$$

i.e., in all three cases we end up, after three β-reductions, with the same normal form, which is the variable w.

A simple expression that has no normal form is the self-application

$$(\lambda u.(u\ u)\ \lambda u.(u\ u)) \rightarrow_\beta (\lambda u.(u\ u)\ \lambda u.(u\ u)) \rightarrow_\beta \ldots\ ,$$

that incessantly reproduces itself.

This self-application may be used to construct a simple expression whose reduction, depending on the order in which redices are contracted, may or may not terminate. Looking at the application

$$((\lambda u.\lambda v.u\ \lambda w.w)\ (\lambda u.(u\ u)\ \lambda u.(u\ u)))\ ,$$

we immediately recognize that the abstraction $\lambda u.\lambda v.u$ in the operator position is a selector function that reproduces its first argument, i.e., $\lambda w.w$, but throws away the second one, which in this particular case is the self-application. Thus, when doing outermost redices first, we get:

$$((\lambda u.\lambda v.u\ \lambda w.w)\ (\lambda u.(u\ u)\ \lambda u.(u\ u))) \rightarrow_\beta (\lambda v.\lambda w.w)\ (\lambda u.(u\ u)\ \lambda u.(u\ u)) \rightarrow_\beta \lambda w.w\ ,$$

i.e., the sequence terminates after two steps, with the first argument as the normal form. However, if we do the innermost redices first, i.e., the operands of the application, we get:

$$((\lambda u.\lambda v.u\ \lambda w.w)\ (\lambda u.(u\ u)\ \lambda u.(u\ u))) \rightarrow_\beta ((\lambda u.\lambda v.u\ \lambda w.w)\ (\lambda u.(u\ u)\ \lambda u.(u\ u))) \rightarrow_\beta \ldots\ .$$

The self-application keeps reproducing itself forever, and the entire application never reaches its normal form.

The problem in this and all other cases where we have reduction sequences that do not terminate, even though normal forms do exist, is that they try to reduce subexpressions whose normal forms do not contribute to the normal form of the entire expression but are eventually discarded by selector functions. Such reduction sequences may perform superfluous computations in harmless cases, which should already be reason enough to try to avoid them, but may cause non-termination in worst cases, which renders it imperative to look for sequences that guarantee termination with normal forms if they exist at all.

The strategy that accomplishes this is called normal order reduction (or operands-when-needed). The idea is to apply abstractions to unevaluated operands and to force their reduction if and only if there are normal forms other than abstractions, i.e., variables or applications that cannot be β-reduced, in operator positions of applications.

This strategy may be defined by a transformation function τ_N that maps λ-expressions into λ-expressions as follows:

$$\tau_N(\ e\ ) = \begin{cases} v \quad \text{if} \quad e =_s v \in V \\ \\ \lambda v.\tau_N(\ e_b\ ) \quad \text{if} \quad e =_s \lambda v.e_b \\ \\ \tau'_N(\ e\ ) \quad \text{if} \quad e =_s (\lambda v.e_b\ e_2) \\ \\ \tau'_N(\tau_N(\ e_1\ )\ e_2) \quad \text{if} \quad e =_s (e_1\ e_2) \ \text{ and } \ e_1 \neq_s \lambda v.e_b \end{cases},$$

and for τ'_N we have

$$\tau'_N(\ e\ ) = \begin{cases} \tau_N(\ e_b[\ v \leftarrow e_2\ ]\ ) \quad \text{if} \quad e =_s (\lambda v.e_b\ e_2) \\ \\ (e_1\ \tau_N(\ e_2\ )) \quad \text{if} \quad e =_s (e_1\ e_2) \ \text{ and } \quad e_1 \neq_s \lambda v.e_b \end{cases}.$$

The function τ_N in fact defines an abstract evaluator for expressions of the pure λ-calculus, similar to the evaluator EVAL that we introduced in Sect. 3.2 to define the evaluation of AL programs. Unlike EVAL, τ_N is recursively driven down just the operator expressions of an application, but it does not touch the operand expression until the operator is done and turns out not to be an abstraction (the last case in the definition of τ'_N). If it is an abstraction, then the operand is substituted for free occurrences of the λ-bound variable as it is (the first case in the definition of τ'_N).

Normal order reduction is also referred to as being outermost-leftmost, meaning that it always reduces the redex that is not contained in any other redex and that the abstractor involved is to the left of the abstractors of all other redices. This strategy is also known as call-by-name in conventional programming languages such as ALGOL or SIMULA.

Unfortunately, this strategy, though guaranteeing termination with normal forms if they exist, does not come without a penalty. If the operand expression is substituted in more than one place in the abstraction body, there is a good chance that there will be multiple reductions of the same thing. However, as we will see later, this redundancy can be avoided by a clever implementation technique called lazy evaluation which ensures that operand expressions are reduced at most once and only to the extent absolutely necessary to arrive at normal forms.

The strategy that may cause the termination problems discussed above is called applicative order reduction. It forces the reduction to normal forms of both the operator and the operand expression of an application before the application itself is reduced, though with one exception: reductions do not penetrate abstractions that are in operator positions. It is defined by a transformation function τ_A on λ-expressions as

$$\tau_A(\, e\,) = \begin{cases} v \quad \text{if} \;\; e =_s v \in V \\ \\ \lambda v.\tau_A(\, e_b\,) \quad \text{if} \;\; e =_s \lambda v.e_b \\ \\ \tau'_A(\lambda v.e_b \;\tau_A(\, e_2\,)) \quad \text{if} \;\; e =_s (\lambda v.e_b \; e_2) \\ \\ \tau'_A(\tau_A(\, e_1\,)\;\tau_A(\, e_2\,)) \quad \text{if} \;\; e =_s (e_1\; e_2) \;\text{ and }\; e_1 \neq_s \lambda v.e_b \;\; , \end{cases}$$

and for τ'_A we have

$$\tau'_A(\, e\,) = \begin{cases} \tau_A(\, e_b[\, v \leftarrow e_2\,]) \quad \text{if} \;\; e =_s (\lambda v.e_b \; e_2) \\ \\ (e_1\; e_2) \quad \text{if} \;\; e =_s (e_1\; e_2) \;\text{ and }\; e_1 \neq_s \lambda v.e_b \;\; . \end{cases}$$

It may be noted that this is the operands-first strategy of the abstract evaluator EVAL introduced in Sect. 3.2, though with two exceptions: EVAL does not evaluate the alternatives of `if_then_else` clauses before the selection has been made, and it also includes measures that prevent runaway recursions when `letrec` expressions are evaluated.

These are the cases where the applicative order regime must be compromised in order to avoid termination problems at least with ordinary (mutually) recursive functions that would terminate safely under a normal order regime, provided they include proper termination conditions.

With these exceptions, applicative order reduction is a perfectly acceptable strategy that is fairly easy to implement mechanically and is also fast. It evaluates operands exactly once and to normal forms irrespective of need, and may fail to terminate only in some exotic cases, e.g., with self-applications of functions as operands. Under the name call-by-value this strategy dominates the scene in conventional programming languages.

Applicative order reduction is also called innermost-leftmost as it reduces the redex that contains no other redices, and among those it does the one with the leftmost abstractor involved first.

The most important property of sequences of β-reductions is captured by the well-known Church-Rosser theorem. This theorem essentially says that irrespective of the order in which β-reductions are performed on some λ-expression, there exists always a λ-expression in which two different sequence of β-reductions may be joined again. This may be formalized as follows:

Let e_0, e_1, e_2, e_3 be λ-expressions, then
$e_0 \longmapsto_\beta e_1$ and $e_0 \longmapsto_\beta e_2$ implies that there exists an e_3 such that
$e_1 \longmapsto_\beta e_3$ and $e_2 \longmapsto_\beta e_3$.

The expressions e_1 and e_2 need not necessarily be syntactically different, i.e., they may be α-convertible into each other, and e_3 need not necessarily be a normal form since none may exist. However, if it is a normal form then it is guaranteed to be unique. Put another way, the normal form of a λ-expression is invariant against the reduction sequences by which it can be reached.

The proof of this theorem is rather complex, covering several pages, but there is a very plausible explanation that may help us to understand why the theorem makes sense.

Disregarding for a moment the name-clashing problem, β-reduction is a very simple operation: it performs a context-free substitution of equals by equals, i.e., it replaces a syntactical figure – an application – by another one without causing any (side) effects somewhere else in a larger expression (or context). The replacement depends solely on the components of the application and is always the same, irrespective of the context in which the substitution takes place – a property that is also referred to as referential transparency. Moreover, the syntax of λ-expressions ensures that β-redices are non-overlapping: they may be fully nested inside or completely independent of each other but they do not share any components. Hence the determinacy of normal forms, apart from termination problems, is bound to be invariant against the order in which β-reductions are performed.

Allowing name clashes does not change anything. The classical definition of the β-reduction rule deals with them just by renaming the bound variables that engage in the clashes, but otherwise leaves the abstractions as they are. Once names have been changed to take abstractions out of conflicts, β-reduction is again a simple substitution operation as above. What matters in this game is that the binding structures of the abstractions are preserved; the choice of variable names by which this is accomplished is irrelevant.

When the nameless Λ-calculus introduced in Sect. 4.4 is used instead, the binding structures are preserved by other, more elaborate means, but it has been proven that the Church-Rosser property, or confluence, holds as well.

Besides normal forms that are the ultimate goal of the process of β-reducing expressions of the pure λ-calculus, there are two important intermediate variants of normal forms. In order to distinguish the three of them, we refer to a λ-expression as being a

- full normal form if it contains no β-redices (instead of normal form, whenever ambiguities may otherwise occur), and the λ-calculus that computes full normal forms is said to be fully normalizing (or just normalizing);
- weak (head) normal form if it is a top-level abstraction (which may contain redices in its body) or a top-level application of an n-ary abstraction to fewer than n operands that are in weak normal form, and the λ-calculus that computes just weak normal forms is said to be weakly normalizing;
- head normal form if it is a special top-level abstraction

$$\lambda u_1 \ldots \lambda u_n.(\ldots(\ u_i\ e_1\ )\ldots e_m\ ) \quad \text{with } i \in \{0, \ldots,\ n-1\}$$

that cannot change its shape anymore to the left of the head variable u_i since further β-reductions can take place only in the operand expressions $e_1, \ldots,\ e_m$; the λ-calculus that computes head normal forms is said to be head normalizing.

These three normal forms are obviously related to each other as follows: every λ-expression that is in (full) normal form is also in head normal form, and every expression in head normal form is also in weak head normal form, but not the other way around, i.e., they form a hierarchy. Full normalization and head normalization both require full β-reductions as they may necessitate substitutions and reductions under abstractors, which could cause name clashes. Naive substitutions suffice to weakly normalize since other than top-level reductions, including those of partial applications, are ruled out, which precludes name clashes.

We will see later on that abstract machines for functional languages that are based on what is called compiled graph reduction are just weakly normalizing, and that head normalization is a most suitable strategy to efficiently compute full normal forms, employing full-fledged β-reductions.

4.6 Recursion in the λ-Calculus

The concept of recursion is absolutely essential for the specification of nontrivial algorithms that need to repeat computational steps depending on actual parameter values. In fact, recursive functions are the essence of intuitively computable functions.

We remember that algorithms using recursive functions may in the language AL of Chap. 3 be specified as

$$\texttt{letrec}\ f = \texttt{lambda}\ u_1\ \ldots\ u_n\ \texttt{in}\ \dashv \ldots\ f\ \ldots\ f\ \ldots \vdash\ \texttt{in}\ (f\ a_1\ \ldots\ a_n)$$

(the symbols $\dashv$ and $\vdash$ are used as delimiters for the function body expression).

Following the definition of the abstract evaluator EVAL, the first step executed by this algorithm consists of substituting the single occurrence of the

function identifier f in the goal expression by the right-hand side of the defining equation. All occurrences of f in this right-hand side are, in turn, substituted by the entire `letrec` construct which, however, has the goal expression replaced by the identifier f, resulting in the application

$$(\texttt{lambda}\, u_1 \ldots u_n\ \texttt{in}\ \dashv \ldots \texttt{letrec}\, f = \texttt{lambda}\, u_1 \ldots u_n\ \texttt{in}\ \dashv \ldots \vdash\ \texttt{in}\, f \\ \ldots\ \texttt{letrec}\, f = \texttt{lambda}\, u_1 \ldots u_n\ \texttt{in}\ \dashv \ldots \vdash\ \texttt{in}\, f \ldots \vdash\ a_1 \ldots a_n)\ .$$

This mechanism replicates the `letrec` construct in all the places in which it is needed for subsequent recursive calls. Its goal expression is just the identifier f of the function that is to be called, which in fact serves as the selector for the right-hand side of the function definition. It is important to note here that the evaluation of this construct, i.e., the actual selection of the abstraction, is postponed until the evaluator EVAL is driven right in front of it.

Since the λ-calculus does not know defining equations, we have to find another way of representing the concept of reproducing expressions in themselves, which is what recursion apparently is all about. We might try to accomplish this with self-applications, of which we know one special case already, the expression $(\lambda u.(u\ u)\ \lambda u.(u\ u))$.

A self-applicative representation of the above algorithm in the λ-calculus that would immediately come to mind is this:

$$(\ldots((f^*\ f^*)\ a_1)\ldots a_n)\ ,$$

$$\text{where}\ \ f^* =_s \lambda f.\lambda u_1 \ldots \lambda u_n.\ \dashv \ldots (f\ f) \ldots (f\ f) \ldots \vdash\ .$$

Note that f^* is meant to be syntactically equivalent to the abstraction on the right of the $=_s$ sign; this is not a defining equation, i.e., we have in fact an application

$$(\ldots((\lambda f.\lambda u_1 \ldots \lambda u_n.\ \dashv \ldots (f\ f) \ldots (f\ f) \ldots \vdash\ f^*)\ a_1) \ldots a_n)\ ,$$

which by a first β-reduction step is transformed into

$$(\ldots(\lambda u_1 \ldots \lambda u_n.\ \dashv \ldots (f^*\ f^*) \ldots (f^*\ f^*) \ldots \vdash\ a_1) \ldots a_n)\ .$$

This step reproduces the self-application of f^* twice in the abstraction body, after which the remaining abstraction, beginning at the abstractor λu_1, may be applied to the operand terms $a_1, \ldots, a_n$.

This approach certainly does the job of calling a function recursively in itself, but the entire construction is function-specific: it requires that the recursive calls be specified as self-applications of abstractions that carry an additional parameter to pass along the abstractions themselves.

However, since this is not a very elegant solution, we may wish to look for a suitable universal recursion operator, say s, that, when applied to an abstraction of the general form

$$f =_s \lambda z. \dashv \ldots z \ldots z \ldots \vdash \ ,$$

produces the sequence of β-reductions

$$(s\ f) \longmapsto_\beta (f\ (s\ f)) =_s (\lambda z. \dashv \ldots z \ldots z \ldots \vdash\ (s\ f)) \\ \rightarrow_\beta \dashv \ldots (s\ f) \ldots (s\ f) \ldots \vdash \ ,$$

which reproduces the application $(s\ f)$ in f.

Since the expressions of this sequence are semantically equivalent, we may also write

$$(s\ f) = (f\ (s\ f)) =_s (\lambda z. \dashv \ldots z \ldots z \ldots \vdash (s\ f)) \\ = \dashv \ldots (s\ f) \ldots (s\ f) \ldots \vdash \ .$$

Reading the first equation from right to left, i.e., as $(f\ (s\ f)) = (s\ f)$, we immediately recognize that $(s\ f)$ is a fixed point of f: it is an expression that f maps into itself. We call a closed λ-expression s (whose set of free variables is empty) that satisfies this equation for all λ-expressions f a fixed-point combinator.

Such a combinator may be realized by the self-application

$$Y =_s (p\ p) \quad \text{where} \quad p =_s \lambda u.\lambda v.(v\ ((u\ u)\ v)) \ .$$

We may convince ourselves that this is the case by reducing the application

$$(Y\ f) =_s ((\lambda u.\lambda v.(v\ ((u\ u)\ v))\ p)\ f)$$

in three steps as follows:

$$((\lambda u.\lambda v.(v\ ((u\ u)\ v))\ p)\ f) \rightarrow_\beta \\ (\lambda v.(v\ ((p\ p)\ v))\ f) \rightarrow_\beta \\ (f\ ((p\ p)\ f)) =_s (f\ (Y\ f)) \ .$$

The beauty of this Y-combinator of the λ-calculus is that it takes the self-applications out of individual functions and, at the expense of a few more β-reductions, effects recursive calls on all abstractions in a uniform way.

Consider, as an example, the defining equation

$$f = \lambda u.\lambda v.(((f\ u)\ v)\ ((f\ v)\ u))$$

for a recursive abstraction. We can rewrite the right-hand side of this equation as a semantically equivalent application

$$f = (\lambda z.\ \underbrace{\lambda u.\lambda v.(((z\ u)\ v)\ ((z\ v)\ u))}_{e_b}\ f)$$

or, when using e_b as an abbreviation for the abstraction body, as:

$$f = (\lambda z.e_b\ f)\ .$$

To give this equation the same form as $(Y\ f) = (f\ (Y\ f))$, a solution for f must obviously look like

$$f =_s (Y\ \lambda z.e_b)\ ,$$

so that we get

$$(Y\ \lambda z.e_b) = (\lambda z.e_b\ (Y\ \lambda z.e_b))\ .$$

That is to say, we can take a defining equation for a recursive function and turn it into a recursive λ-expression by (1) abstracting occurrences of the function identifier out of the abstraction on the right-hand side of the equation and (2) applying to this new abstraction the Y-combinator – a very simple recipe indeed.

Unfortunately, the Y-combinator is not suited for applicative order reductions since the application $(Y\ f)$ would incessantly reproduce itself in operand position, generating the unending sequence of expressions

$$(Y\ f) \longmapsto_\beta (f\ (Y\ f)) \longmapsto_\beta (f\ (f\ (Y\ f))) \longmapsto_\beta \ldots$$

This may be prevented either by switching to a normal order regime just for applications of the Y-combinator, or by introducing, as an extension of the λ-calculus, a primitive recursion operator μ. Using this operator, we may replace $(Y\ \lambda z.e_b)$ by $\mu z.e_b$, which binds the variable z recursively in e_b and reduces like this:

$$\mu z.e_b \rightarrow_\mu (\lambda z.e_b\ \ \mu z.e_b) \rightarrow_\beta e_b[\ z \leftarrow \mu z.e_b\ ]\ .$$

That is, the operator μ substitutes free occurrences of the variable z in the body expression e_b by (copies of) the entire expression as it is, which is exactly what is needed to prevent runaway recursions, provided that e_b contains proper termination conditions.

Both the Y-combinator and the μ-operator may be employed to represent mutually recursive functions as λ-expressions as well. We simply need to consider the most general case of a set of defining equations

$$\begin{aligned} f_1 &= \ldots f_1 \ldots f_i \ldots f_k \ldots \\ &\ldots \\ f_i &= \ldots f_1 \ldots f_i \ldots f_k \ldots \\ &\ldots \\ f_k &= \ldots f_1 \ldots f_i \ldots f_k \ldots \end{aligned}$$

in which each function recursively calls each other function of the set. These equations may be rewritten as

$$f_i = (\lambda \bar{z}.e_b^{(i)}\ \ f_1 \ldots f_i \ldots f_k)\ ,$$

where

$$\lambda\bar{z}.e_b^{(i)} =_s \lambda z_1 \ldots \lambda z_i \ldots \lambda z_k. \dashv \ldots z_1 \ldots z_i \ldots z_k \ldots \vdash$$

for all $i \in \{1, \ldots, k\}$. The expressions $e_b^{(i)}$ are generally abstractions themselves that need to be applied to operands (which are not included here).

If we now abstract once more the variable f_i we get

$$f_i = (\lambda z_i.(\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots z_i \ldots f_k) \ f_i) \ ,$$

and, when introducing the Y-combinator, we obtain as a solution of this equation:

$$f_i = (Y \ \lambda z_i.(\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots z_i \ldots f_k)) \ .$$

Instead of the Y-combinator, we may alternatively use the primitive recursion operator μ, in which case we get:

$$f_i = \mu z_i.(\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots z_i \ldots \ f_k) \ .$$

β-reduction of the Y-combinator version or direct reduction of the μ-operator version yields

$$f_i = (\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots \underbrace{(Y \ \lambda z_i.(\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots z_i \ldots f_k))}_{f_i} \ldots f_k)$$

or

$$f_i = (\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots \underbrace{\mu z_i.(\lambda\bar{z}.e_b^{(i)} \ f_1 \ldots z_i \ldots f_k)}_{f_i} \ldots f_k) \ ,$$

respectively.

These reduction steps are the λ-calculus equivalents of selecting functions from `letrec` expressions, as specified in Sect. 3.2. We remember that occurrences of function identifiers in their goal expressions are substituted by the abstractions on the right-hand sides of their defining equations, and that all recursive occurrences of function identifiers in these abstractions are substituted by copies of the full `letrec` expressions, with just the particular identifiers as their goal expressions.

The equivalent of reproducing the `letrec` expression is accomplished here by replacing a recursive call of a function f_i with an application of the abstraction $\lambda\bar{z}.e_b^{(i)}$ to the full set of (recursive) λ-abstractions for repeated use in $e_b^{(i)}$.

We can also turn this argument around and say that the way we need to realize mutually recursive functions in the clean setting of the pure λ-calculus confirms that we did the right things in the evaluation of `letrec` expressions by the abstract evaluator EVAL.

4.7 A Brief Outline of an Applied λ-Calculus

The pure λ-calculus, though providing a precise formal model of computable functions, is of course not suited as a language for the specification of real-life algorithmic problems. What is missing are the usual representations of numbers, truth values, character strings and data-structuring facilities, together with sets of primitive operators (functions) defined on them, as they are available in full-fledged programming languages.

These items may be included in a set C of constant expressions.

Applications of primitive functions such as $+$, $-$, ..., `gt`, `eq`, ..., `and`, `or`, `not`, ... to legitimate arguments are called δ-redices, and their evaluation is referred to as δ-reductions or δ-contractions. Legitimate arguments are those on which the primitive functions are defined, i.e., numbers in the case of arithmetic functions, numbers or character strings in the case of relational functions, and Boolean values in the case of logic functions.

Let e and e' be expressions of an applied λ-calculus, then e is said to be δ-reducible to e' in n steps, denoted as $e \rightarrow_{\delta\_n} e'$, iff

- $e =_s (\ldots(pf\ e_1)\ldots e_n)$ and pf is a primitive function of arity n;
- both e and e' do not contain any free variables;
- the expressions $e_1, \ldots, e_n$ do not contain any β- or δ-redices;
- the function is defined on the values of the expressions $e_1, e_2, \ldots, e_n$ or, to put it another way, these values are within the function's domain.

This definition in fact demands that δ-reductions be performed after all β-reductions affecting free occurrences of bound variables in e have been done, and the arguments of the applications are subsequently reduced to values. Otherwise it cannot be decided whether or not the applications are δ-reducible.[8]

`If_then_else` clauses have in an applied λ-calculus the form $((e_0\ e_1)\ e_2)$, where e_0, e_1, e_2 denote the predicate, consequent and alternative expressions, respectively. It is assumed here that the Boolean values to which the predicate is expected to reduce are the combinators

$$\texttt{true} \equiv K =_s \lambda u.\lambda v.u \qquad \texttt{false} \equiv \overline{K} =_s \lambda u.\lambda v.v \ ,$$

which may be considered primitive selector functions. Again, we need to make an exception here to the general rule of reducing the arguments before applying the primitive functions to them: the combinators K and $\overline{K}$ must be applied to their arguments as they are in order to prevent recursive functions with proper termination conditions from falling nevertheless into the black hole of runaway recursions.

As an example of an algorithm specified in an applied λ-calculus, we consider again the factorial function (see also Sect. 2.1). This function takes the syntactical form

[8] Exceptions to this definition are primitive list-processing functions such as `first`, `rest`, `empty`, etc. which demand that their arguments be lists, but the list components may contain β- and δ-redices.

$$fac =_s \lambda n.((((\texttt{gt}\ 1)\ n)\ ((*\ (fac\ ((-\ 1)\ n)))\ n))\ 1)\ .$$

By abstraction of the recursively defined variable fac we get

$$fac =_s (\lambda f.\lambda n.((((\texttt{gt}\ 1)\ n)\ ((*\ (f\ ((-\ 1)\ n)))\ n))1)\ fac)\ ,$$

i.e., fac is a fixed point of the abstraction

$$\lambda f.\lambda n.((((\texttt{gt}\ 1)\ n)\ ((*\ (f\ ((-\ 1)\ n)))\ n))\ 1)\ ,$$

and we obtain as the Y-combinator solution for fac:

$$fac =_s (Y\ \lambda f.\lambda n.((((\texttt{gt}\ 1)\ n)\ ((*\ (f\ ((-\ 1)\ n)))\ n))\ 1))\ .$$

The language AL of Chap. 3 is just a syntactical variant of an applied λ-calculus. It merely takes a few minor modifications of the above representation of factorial to turn it into an expression of this language.

First of all, we can save a few of those messy parentheses by uncurrying the nested applications to get:

$$fac =_s (Y\ \lambda f.\lambda n.((\texttt{gt}\ 1\ n)\ (*\ (f\ (-\ 1\ n))\ n)\ 1))\ .$$

Next, we add a little syntactic sugar by turning the abstraction body into an `if_then_else` construct:

$$fac =_s (Y\ \lambda f.\lambda n.\texttt{if}\ (\texttt{gt}\ 1\ n)\ \texttt{then}\ (*\ (f\ (-\ 1\ n))\ n)\ \texttt{else}\ 1)\ .$$

Finally, we replace the Y-combinator by an equivalent `letrec` construct (with the goal expression left open) to obtain:

$$\texttt{letrec}\ fac = \lambda n.\texttt{if}\ (\texttt{gt}\ 1\ n)\ \texttt{then}\ (*\ (fac\ (-\ 1\ n))\ n)\ \texttt{else}\ 1\ \texttt{in}\ \ldots\ .$$

Semantic equivalence between an applied λ-calculus and AL can be obtained by simply declaring the substitutions carried out by the abstract evaluator EVAL to be, conceptually, β-reductions.

4.8 Overview of a Typed λ-Calculus

The λ-calculus is a functional model of computation that deals primarily with the operational aspects of transforming applications of functions to arguments into function values. The Church-Rosser property guarantees that these applications have unique values (or normal forms), if they exist at all, i.e., the functions map, as required, the same argument values into the same function values.

The λ-calculus that we have considered so far performs these transformations in a most general way, as it simply maps λ-expressions into λ-expressions. However, we have also seen that in the language AL, which in fact is an applied

λ-calculus, we may specify a lot of syntactically correct expressions that make little or no sense semantically. Their evaluation may terminate with expressions that still include applications that are neither β- nor δ-reducible. Typical examples are applications of primitive functions to incompatible arguments, e.g., of arithmetic operators to something other than numbers, or applications with something other than abstractions or primitive functions, say numbers or free variables, in operator positions.

If such expressions are not what we expect as the results, then we could decide to consider them erroneous, take corrective actions to deal with the likely cause of the error in the initial expressions, and try again. This kind of debugging has the advantage of taking place entirely in the domain of expressions, and not on the level of compiled code and register contents of which we do not (want to) know anything when writing high-level algorithms. However, the trouble with this approach is that it is of only limited practical value. Even fairly simple recursive algorithms tend to unfold to intermediate expressions of considerable size and complexity, in which the causes of irreducible applications are hard to track down.

Alternatively, we could try to make sure beforehand and by separate means that all applications in a given expression are β- or δ-reducible, i.e., the expression can be reduced to a normal form that does not contain any leftover applications. This is equivalent to restricting the freedom in specifying algorithms to those that, loosely speaking, produce meaningful output if fed with meaningful input. Otherwise, the algorithms may simply be rejected as not being executable.

This leads us to the notion of a typed λ-calculus, as opposed to an untyped λ-calculus that we have dealt with so far. It has associated with each expression a set of values that it may legitimately assume; this set of values is said to be the type of the expression. Types introduce into the λ-calculus (and, for that matter, into any programming language) the notion of functions as mappings from sets of legitimate arguments to function values, or from domain sets to range sets.

This typed λ-calculus in fact formalizes what we have said rather casually in Sect. 2.2 about the typing of algorithms.

Typing may be seen as a means to introduce more formal rigor into the design of algorithms and to facilitate reasoning about their correctness. There are good reasons to assume that an algorithm very likely does what it is supposed to do if it is consistently typed. Beyond that, typing is also of considerable practical relevance. Since contemporary computing systems have no built-in type checking facilities, programs written in high-level languages must, at least to some extent, include type declarations to make sure that they do not produce type inconsistencies at runtime, and to aid the compiler in generating type-correct machine code.

4.8.1 Monomorphic Types

Types may be specified by means of type expressions that are basically constructed from atomic types and function types. The atomic types denote sets of numbers (integer or real), Boolean values, characters and character strings, etc., and the function types denote mappings from domain sets to range sets. Unspecified atomic types may be denoted as α, β ... and unspecified function types by, say, $\alpha \to \beta$. We refer to these types as being monomorphic.

Based on these monomorphic types, we may define a typed λ-calculus as follows:

- if v and c respectively denote variables and constants, then $v : \alpha$ and $c : \alpha$ are typed atomic expressions of type α;
- $(e_0 : (\alpha \to \beta)\ e_1 : \alpha) : \beta$ denotes a typed application of type β if e_0 and e_1 are typed λ-expressions of types $(\alpha \to \beta)$ and α, respectively;
- $(\lambda v : \alpha.e_b : \beta) : (\alpha \to \beta)$ is a typed abstraction of type $(\alpha \to \beta)$ if v is a variable of type α and e_b is a typed λ-expression of type β.

Some of the type annotations of abstractions and applications may be dropped if they can in obvious ways be inferred from other types. For instance, if the type of an abstraction is $\alpha \to \beta$, then the type of the λ-bound variable must be α and the type of the body expression must be β.

The type of a curried n-ary abstraction can be easily inferred from

$$\lambda v_1 : \alpha_1.\lambda v_2 : \alpha_2 \ldots \lambda v_{n-1} : \alpha_{n-1}.\lambda v_n : \alpha_n.e_b : \beta$$

as being

$$(\alpha_1 \to (\alpha_2 \to \ldots (\alpha_{n-1} \to (\alpha_n \to \beta)) \ldots)) \ .$$

If no ambiguities can occur, we may drop parentheses from such nested function types and simply write

$$\alpha_1 \to \alpha_2 \to \ldots \alpha_{n-1} \to \alpha_n \to \beta \ ,$$

assuming association to the right.[9]

A typed β-reduction is defined as

$$(\lambda v.e_b : (\alpha \to \beta)\ e_a : \alpha) \to_\beta e_b[\ v : \alpha \leftarrow e_a : \alpha\ ] : \beta \ .$$

It does the same as an untyped β-reduction – substituting the argument expression e_a for free occurrences of v in the abstraction body e_b – but it does so if and only if the type of the argument matches the type of the λ-bound variable (or the domain type of the abstraction), returning an expression of type β. Otherwise, the application does not qualify as a redex.

[9] The types of uncurried n-ary abstractions $\lambda v_1\ \ldots\ v_n.e_b$ are usually denoted as $\alpha_1 * \alpha_2 *\ \ldots\ \alpha_{n-1} * \alpha_n \to \beta$, with the parameter types forming a product type.

It is important to note that in a typed β-redex the expression in operator position must be of a higher type, i.e., containing at least one more type constructor $\rightarrow$ than the expression in operand position. Unfortunately, this has the unpleasant consequence of ruling out self-applications in which both operator and operand are the same and therefore must have the same type. This in turn would mean that we cannot have a Y-combinator and hence, strictly speaking, we would have no recursion. A typed λ-calculus would therefore be inherently strongly normalizing in the sense that every typed λ-expression has a normal form that can be reached after finitely many β-reductions.

Clearly, this cannot at all be a desirable feature since the absence of recursion would severely limit the expressive power of a typed λ-calculus. It would exclude all the interesting computational problems that require repeating sequences of reduction steps depending on actual parameters. What would be left to be computable would be merely rather tedious problems that fall into the same class as polynomials.

Fortunately, the problem with recursions is caused not by typing itself but by its implications with regard to self-applications. It is possible to infer a type for every fixed-point combinator μ that satisfies the equation $(\mu\ f) = (f\ (\mu\ f))$. If we assume the type α for $(\mu\ f)$ and $(\alpha \rightarrow \beta)$ for f, we obtain

$$(\mu\ f) : \alpha = (f : (\alpha \rightarrow \beta)\ (\mu\ f) : \alpha) : \beta \ .$$

We see immediately what needs to be done to make the types on both sides of the equation the same, or to unify them. If we set $\alpha = \beta$ we get

$$(\mu\ f) : \alpha = (f : (\alpha \rightarrow \alpha)\ (\mu\ f) : \alpha) : \alpha \ .$$

We can now conclude that if the type of f is $\alpha \rightarrow \alpha$ then the type of μ must be $((\alpha \rightarrow \alpha) \rightarrow \alpha)$ for the type of $(\mu\ f)$ on the left-hand side of the equation to be α.

So, a typed λ-calculus must obviously be extended by a primitive recursion operator μ as introduced in Sect. 4.6, in order to get around the typing problem with the Y-combinator.

The types of the primitive binary arithmetic, relational and logic functions are of the general form $\alpha \rightarrow \alpha \rightarrow \beta$. If *num*, *char* and *bool* respectively denote the types of numbers, characters (character strings) and Boolean values, we have for

- the arithmetic functions $+$, $-$, $*$, $/$ the type $num \rightarrow num \rightarrow num$;
- relational functions such as `lt`, `eq`, ... either the type $num \rightarrow num \rightarrow bool$ or the type $char \rightarrow char \rightarrow bool$;
- logic functions such as `and`, `or`, ... the type $bool \rightarrow bool \rightarrow bool$.

Assuming, for the sake of simplicity, that the expressions in a list all have the same type, say α, and that the type of the entire list can then be denoted as $list\_of\ \alpha$, we have for the list functions

- `empty` the type $list\_of\ \alpha \rightarrow bool$;

- `first` the type $list\_of\ \alpha \to \alpha$;
- `rest` the type $list\_of\ \alpha \to list\_of\ \alpha$;
- `append` the type $list\_of\ \alpha \to list\_of\ \alpha \to list\_of\ \alpha$.

Applications of these functions constitute δ-redices if and only if the arguments are of the required types, otherwise the applications are not δ-reducible.

4.8.2 Polymorphic Types

The monomorphic types that we have discussed so far are not too well suited for abstractions that apply uniformly to all argument expressions irrespective of their types. Typical examples are the recursion operator μ and the abstraction $\lambda v.v$ that respectively are supposed to duplicate and reproduce any λ-expression that comes along as an argument, or the abstraction $\lambda u.\lambda v.u$ that is supposed to reproduce any first and to drop any second argument expression. Also, if we were to write, say, a function that sorts the elements of a list, then we would want this function to be applicable to lists of any element type.

To deal with such functions in a typed λ-calculus, we need to introduce the more general concept of polymorphic types (or of type schemata). They are usually denoted as polymorphic type variables (or identifiers) such as $*\alpha$, $*\beta$ that must be read as for all types α, for all types β, respectively.

At first sight, this seems to be equivalent to no typing at all. However, in the context of an applied λ-calculus where we have primitive functions requiring that their arguments be of a particular monomorphic type, polymorphic typing makes a lot of sense. We can set out typing the λ-bound variables of a uniformly applicable abstraction polymorphically, and then instantiate the type variables consistently with monomorphic types, depending on the context in which the abstraction is actually used.

Consider as an example again the function *twice* introduced in Sect. 2.1, which in λ-calculus notation is given by

$$twice =_s \lambda f.\lambda v.(f\ (f\ v))\ .$$

This function is supposed to accept as its first argument any unary function and to apply it twice to an argument of compatible type. To find the most general (polymorphic) type of *twice*, we assume that the variable v has some polymorphic type $*\alpha$ and that the application $(f\ v)$ has type $*\beta$, which means that f must be of type $*\alpha \to *\beta$ for this application to be consistently typed. Looking now at the application $(f\ (f\ v))$, we see that if f is of type $*\alpha \to *\beta$ and the argument $(f\ v)$ is of type $*\beta$, then the types $*\alpha$ and $*\beta$ must obviously be the same for this to work out. So, we conclude that f must be, say, of type $*\alpha \to *\alpha$ and v must be of type $*\alpha$, which must also be the type of the function value.[10] So, we get as the type of *twice*

[10] We could, of course, have chosen the type variable $*\beta$ instead of $*\alpha$ here, which would not have made any difference.

$$twice : (*\alpha \rightarrow *\alpha) \rightarrow *\alpha \rightarrow *\alpha = \lambda f : (*\alpha \rightarrow *\alpha).\lambda v : *\alpha.(f\ (f\ v)) : *\alpha\ .$$

If we now plan to apply *twice* to the function *square* and the value 2, we need to infer a most general type for

$$square =_s \lambda u.(*\ u\ u)$$

first, which is fairly easy: we know that the primitive function $*$ is of type $num \rightarrow num \rightarrow num$, so both the variable u and the function value computed by the expression $(*\ u\ u)$ must be of type num, i.e., the type of *square* is given by

$$square : num \rightarrow num\ =\ \lambda u : num.(\ *\ u\ u\ ) : num\ .$$

Looking now at the application

$$((twice : (*\alpha \rightarrow *\alpha)\ \rightarrow *\alpha \rightarrow *\alpha\ \ square : num \rightarrow num)\ 2 : num)\ ,$$

we see immediately that we have a consistently typed application if the polymorphic type $*\alpha$ of *twice* is instantiated (or unified) with the monomorphic type *num*.

In another context, we may wish to apply *twice* to the list function `rest` and to a list of type $list\_of\ *\beta$, in which case `rest` must be of the polymorphic type $list\_of\ *\beta \rightarrow list\_of\ *\beta$. Consistent typing of the application

$$((twice : (*\alpha \rightarrow *\alpha) \rightarrow *\alpha \rightarrow *\alpha\ \texttt{rest} : list\_of\ *\beta \rightarrow list\_of\ *\beta)\ llist : list\_of\ *\beta)$$

(where *llist* is a placeholder variable for a list) then obviously requires that the polymorphic type $*\alpha$ be instantiated (unified) with the polymorphic type $list\_of\ *\beta$. Proper instantiation of the type $*\beta$, in turn, may depend upon type unifications in a larger context of this application.

Type annotations on λ-bound variables have scopes that coincide with the respective binding scopes, i.e., they extend over the body expressions of the respective abstractions. All occurrences of the same polymorphically typed variable within a particular scope must be instantiated with the same type, which may be monomorphic, as in the case of *twice* applied to *square* and 2, or another polymorphic type, as in the case of applying *twice* to `rest` and a list of type $list\_of\ *\beta$.

The rules for consistent type annotations are defined by a type system. These rules in fact specify a type checker or a type inference system that may be used to determine whether or not an expression is (or can be) consistently typed.

An expression (or a program) is said to be well typed or safe if no type inconsistencies can occur when evaluating (reducing) it. A type system is sound if every expression that is typed according to its rules is safe. If the type system is sound, then all expressions can be fully type-checked (or types can be inferred) before they are actually evaluated. Languages with sound type

systems are said to be statically typed, as opposed to languages like AL that are said to be dynamically typed since type checking is done while evaluating an expression (or executing a program).

Type checking assumes that expressions (or programs) are fully type-annotated by the user and need only be checked for type consistency. This is the case with all conventional programming languages such as PASCAL, FORTRAN, C, C^{++} or JAVA that are monomorphically typed and require that all variables be declared with respect to their types.

Type inference is the more sophisticated approach taken in almost all functional languages, e.g., HASKELL, CLEAN or ML. It is based on polymorphic type systems. Ideally, the user need not be concerned with type annotations at all as they may be fully inferred from the types of primitive functions and of constant values, as exemplified by means of the functions *twice* and *square*, or from argument types. There are only a few odd cases where type annotations on abstractions are necessary to overcome ambiguities.

It should be noted here that typing, though generally considered very helpful in writing algorithms dedicated to a specific purpose (solving a specific problem), may also cause some rather unpleasant problems that do not occur in untyped languages. Other than ruling out self-applications, there are certain expressions that cannot be consistently typed and are therefore rejected by the type system.

An example is again an application of the function *twice*, this time to the primitive function `first` and to a list of type $list\_of\ *\beta$. Since `first` is of type $list\_of\ *\beta \rightarrow \beta$, it cannot be unified with the argument type $*\alpha \rightarrow *\alpha$ of *twice* and is therefore rejected, even though this application makes perfect sense if $*\beta = list\_of\ \gamma$, i.e., the components of the list are lists themselves.

Other typing problems are caused by applications of the same λ-bound variable in different contexts. For instance, the type of the abstraction

$$\lambda f.\lambda u.\lambda v.(f\ u(f\ u\ v))$$

can be inferred as

$$(*\alpha \rightarrow *\beta \rightarrow *\beta) \rightarrow *\alpha \rightarrow *\beta \rightarrow *\beta\ ,$$

which means that f must be of type $*\alpha \rightarrow *\beta \rightarrow *\beta$, u must be of type $*\alpha$ and v must be of type $*\beta$ for a function value of type $*\beta$.

However, if this abstraction is slightly modified to

$$\lambda f.\lambda u.\lambda v.(f\ u\ (f\ v))\ ,$$

the type system fails for it tries unsuccessfully to infer a type $*\alpha \rightarrow *\beta \rightarrow *\gamma$ for the first occurrence of f and a type $*\alpha \rightarrow *\beta \rightarrow *\gamma$ for the second occurrence of f in the abstraction body, and to unify both types. The problem here is that f is assumed to be both a unary and a binary function.

This excursion into typing was intended to bring out the conceptual differences between an untyped and a typed λ-calculus and its implications for the

design of algorithms. However, typing has little or no bearing on the various abstract machines that we will discuss in the remainder of this text, as the basic mechanisms and runtime structures supported by these machines remain the same irrespective of whether they execute typed or untyped languages.

4.9 Summary

The λ-calculus provides the theoretical foundations of algorithmic programming and program execution. It is a theory of computable functions that deals with operators, their application to operands, and with the systematic construction of complex operators (or algorithms) from simpler ones. Most importantly, it clearly defines the role of variables in this game, specifically the substitution of variables by expressions, and variable scoping.

The essence of this theory is captured in the pure λ-calculus. Its expressions are composed of variables, abstractions (of variables from expressions), and applications of operator to operand expressions, and it knows only one transformation rule called β-reduction. This rule specifies the result of applying an abstraction to a legitimate argument expression as its substitution for free occurrences of the bound variable in the abstraction body. The difficult part of this rule is due to potential naming conflicts between occurrences of free variables in the argument and bound variables in the abstraction body. These conflicts need to be correctly resolved in order to guarantee the determinacy of results irrespective of execution orders. The classical solution takes the bound variable out of the conflict by α-conversion into another variable name. An alternative solution consists in a dynamic indexing scheme for bound variables that keeps their binding status invariant against β-reductions without changing the variable names. A special variant of this indexing scheme is a nameless Λ-calculus in which bound variables are consistently replaced by indices as placeholders for things to be substituted.

The purpose of performing β-reductions is to reduce λ-expressions step by step to their normal forms, i.e., to expressions that contain no more β-redices. There are basically two strategies for performing sequences of β-reductions. The applicative order regime demands that the operands of applications be reduced to normal forms before abstractions in operator positions are applied to them. This appears to be the most economical strategy as it evaluates operand expressions exactly once, and for good reasons it dominates the scene in conventional programming languages. The normal order regime applies abstractions to operands in unevaluated form, which may cause some redundancy due to the evaluation of multiple copies of the same expressions in different places of substitution in abstraction bodies. However, as the name indicates, this strategy guarantees that normal forms can be reached after finitely many β-reductions, if they exist. This contrasts with applicative order evaluation that may get trapped in runaway recursions in subexpressions that do not contribute to normal forms.

The Church-Rosser property of the λ-calculus ensures that normal forms, apart from the termination problem, are invariant against the order in which β-reductions are performed.

Recursion is in the pure λ-calculus realized by means of the Y-combinator. When applied to an abstraction f, it effects the transformation

$$(Y\ f) \to (f\ (Y\ f))$$

that reproduces the application as argument of f. The trouble with the Y-combinator is that it is realized as a self-application of another abstraction that, under an applicative order regime, has the unpleasant property of incessantly reproducing itself. This problem may be avoided either by switching to a normal order regime or by adding to the pure λ-calculus a special recursion operator μ that in fact performs the recursion in normal order.

Extending the pure λ-calculus by primitive functions that operate on numbers, truth values, character strings and lists is a fairly straightforward matter. The ensuing δ-reduction rules simply demand that the arguments of primitive functions be of compatible types, e.g., numbers in the case of arithmetic primitives, for the applications to become reducible.

The idea of a typed λ-calculus is to make sure that expressions are consistently typed in the sense that no type incompatibilities may occur at execution time. The rules for consistent typing are defined by a type system that, in turn, specifies a type checker or a type inference system. Type checking assumes that expressions (or programs) are fully type-annotated by the user and need only be checked for type consistency, which is the case with almost all conventional programming languages. Type inference systems try to infer fully typed expressions from the types of primitive functions, constants and actual argument types, which is what is done in most functional languages.

Unfortunately, consistent typing, though generally thought to raise confidence in the correctness of algorithms, also imposes some restrictions with regard to the freedom of designing them. Other than ruling out self-applications, there are a number of perfectly meaningful expressions that cannot be consistently typed and are therefore rejected by the type system as not being executable.

References

There are two standard textbooks on the λ-calculus, one by Barendregt [Bar84], the other by Hindley and Seldin [HS86], both of which are highly recommended reading to those who wish to get to the bottom of the subject. Short introductions into the λ-calculus may also be found in almost all textbooks on programming and programming languages, particularly those that focus on function-based or functional programming and its implementation, for example [AS85, PeyJ87, FWH92, PvE93, HaMi99], to mention only a few.

The original paper on the λ-calculus by Church is [Chu41]. The nameless λ-calculus was developed independently by Berkling [Ber76, BeFe82] and de-Bruijn [Bru72]. The formal definition in Sect. 4.4 of β-reduction in this calculus has been taken from [Ber94].

Related original work on other models of computability can be found in [Scho24] and [Cur29, Cur36] on combinators, in [Tur37] on Turing machines, in [Kle36] on recursive functions, and in [Post43] on Post's number system. An introduction to the typed λ-calculus is given in [HS86]. The underlying type system is due to Hindley [Hin69] and Milner [Mil78]

5

The SE(M)CD Machine and Others

We begin with the description of abstract computing machines that evaluate expressions of the pure λ-calculus to weak normal forms. This conservative approach allows to implement β-reductions as naive substitutions since no potential naming conflicts between free and bound variable occurrences have to be dealt with. These machines include in a nutshell everything that is absolutely essential to performing algorithmic computations mechanically. All the other abstract machines that we get to know later on are more or less just descendants of these very basic machines.

5.1 An Outline of the Original SECD Machine

The first machine that we are going to talk about is a slightly modified version of the SECD machine invented by Landin in 1962 as an abstract evaluator for λ-expressions that employs an applicative order strategy.

The operating principle of this machine centers around the idea of delayed substitutions. Rather than performing β-reductions as atomic operations, the machine partitions them, as a measure to improve runtime efficiency, into two steps that are distributed over time. When encountering β-redices, it just collects mappings of bound variables to (operand) expressions in a runtime structure called the environment. All substitutions are subsequently done while traversing the body expressions only once in search of bound-variable occurrences.

Closely related to delayed substitutions is the notion of closures. They pair abstractions that generally include occurrences of free variables with environments containing the (evaluated) expressions that may have to be substituted for them later on, whenever it becomes possible and necessary to actually evaluate these closures.

The name of the machine derives from the four runtime structures it uses, which are

- a code structure C that holds in textual form expressions or fragments of expressions in the order in which they need to be evaluated;
- a value stack S onto which the values of (sub)expressions are pushed;
- an environment structure E whose entries associate (or pair) bound variables with the values by which they need to be substituted;
- a dump stack D onto which entire machine states are pushed when entering, and from which they are retrieved when returning from, (naive) β-reductions, respectively.

Complete λ-expressions are initially set up for evaluation in the code structure C; all other structures are initially empty. The machine takes syntactically complete expressions or subexpressions off the top of C, evaluates them and pushes these values onto stack S. Values that become arguments of (naive) β-reductions are removed from S, paired with the respective λ-bound variables, and prepended to the current environment E. Free occurrences of bound variables that pop to the top of C are replaced with copies of the value entries found for them in E and pushed onto S again. Correct binding scopes are adhered to by isolating, upon entering into a β-reduction, the body expression of the abstraction on the code structure C, setting up in E only the entries that belong to this scope, and pushing the remaining parts of the structures S, E, C, together with the current dump D, onto D. The evaluation of this body expression terminates with its value in S (with all bound-variable occurrences substituted by values from the environment), and with an empty code structure C. When arriving at such a configuration, the machine retrieves, as return continuation, the topmost machine state saved on the dump and continues. It terminates with the value of the entire expression as the sole entry in an otherwise empty stack S, and with all other structures empty.

The applicative order regime requires that applications $(e_0\ e_1)$ that pop to the top of C be re-arranged as

$$(e_0\ e_1) \rightarrow e_1 : e_0 : ap \ .$$

This brings the operand expression e_1 to the front, followed by the operator expression e_0, and then by a special applicator symbol ap, with the separation symbol ' :' denoting a linear ordering among these items. The machine computes the values of e_1 and e_0 in this order, pushes them onto S, and the applicator ap, once it appears on top of C, applies the topmost value in S to the value that is underneath.

As for the values that end up in stack S, the machine needs to distinguish between free variables that are their own values, bound variables for which values must be retrieved from the environment E, applications of variables whose values are the applications themselves and of (applications of) abstractions turned closures.

What the SECD machine does in the latter case may be exemplified by means of the application

$$(((\lambda u.\lambda v.\lambda w.((u\ v)\ w)\ v)\ w)\ u) \ .$$

Applying full-fledged β-reductions and using the protection keys introduced in Sect. 4.3 would conceptually reduce this expression in three steps, as shown in Fig. 5.1.

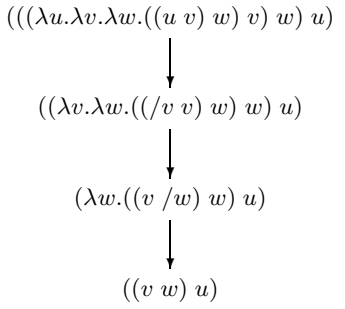


Fig. 5.1. Reducing a nested application that produces name clashes

The dynamic resolution of name clashes by protection keys notwithstanding, we end up with a normal form in which what were originally the bound variables $u,\ v,\ w$ in the abstraction body are now replaced by the free variables $v,\ w,\ u$, respectively, found in the operand positions. Since all protection keys have disappeared again, the β-reductions have in fact been performed as naive substitutions. As already pointed out in Sect. 4.3, this can be done safely if an n-ary abstraction is applied to n arguments and no other abstractions inside the abstraction body need to be penetrated.

The SECD machine uses the concept of closures to postpone substitutions in an abstraction body until all λ-abstractors preceding it have been consumed. As indicated before, a closure is a construct of the form $[\ E\ \lambda v.e_b\ ]$ that pairs an abstraction with an environment, i.e., with a set of variable | value pairs in which $\lambda v.e_b$ must be evaluated. To put it another way, the closure represents the value of $\lambda v.e_b$ in E without actually doing the evaluation.

Using closures, the SECD machine reduces the above application, in principle, as illustrated in Fig. 5.2. [1] It sets out with a closure containing the initial abstraction paired with an as yet empty environment, denoted as an empty list $<>$. This closure is applied as operator to the three operands u, v, w. The first part of β-reducing the innermost redex takes the abstractor λu off the abstraction, consumes the argument v, creates from these two items a variable | value pair $< u\ v >$ and prepends it to the environment list. The next two steps repeat this operation on the remaining β-redices, which builds up the environment $<< u\ v > < v\ w > < w\ u >>$ for the evaluation of the abstraction body $((u\ v)\ w)$ now exposed. In the last step, the substitutions can be carried out safely without worrying about name clashes, and the en-

[1] The notation used in this figure is pure λ-calculus enriched by closures.

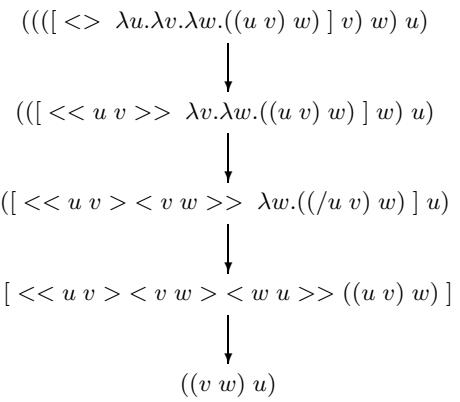


Fig. 5.2. Reducing the nested application of Fig. 5.1 with closures

vironment can subsequently be dropped since everything that completes the β-reductions has been done.

If the same abstraction would be applied to just one or two arguments, in which case we would have partial applications, then the values would be the closures shown in the second and third lines from the top, respectively. In the context of a larger expression they could become the operands of subsequent β-reductions, get into the operator positions of other applications and pick up the missing arguments to become full applications, or be returned as (components of) the value of the entire expression.

Closures are the means that prevent the SECD machine from substituting and performing reductions under abstractors and thus are an essential prerequisite for a more efficient implementation of β-reductions as naive substitutions. The ensuing weak normalization is widely accepted as a good compromise between the amenities of a fully normalizing λ-calculus and simple machinery. It dictates the semantics of almost all functional and function-based programming languages whose implementations return as the results of partial applications the anonymous values `function` or `procedure` (of specific arities corresponding to the number of missing arguments), which are just external representations for closures.[2]

Another problem with the SECD machine arises with applicative order evaluation. As it rules out using the Y-combinator, recursions must be implemented either by adding the special recursion operator μ as introduced in Sect. 4.6, by switching completely to a normal order regime, or by supporting normal order reductions at least as an option.

[2] The semantics of imperative languages take weak normalization one step further in that they completely rule out partial applications, which is primarily a consequence of compiling functions (procedures) to static code that must be fed with full sets of arguments (actual parameters) to execute correctly.

5.2 The SE(M)CD Machine

We will now introduce an improved version of Landin's SECD machine that can do both applicative and normal order reductions, the latter to overcome the Y-combinator problem and also to realize selection among alternatives without evaluating them. It employs a constructor syntax to represent λ-expressions in a form that helps to distinguish between the two evaluation regimes and also lends itself more elegantly to mechanized interpretation. All it takes to have this constructor syntax supported by the machine is another stack M for the temporary storage of constructor symbols.

β-reductions are implemented naively as in the original SECD machine, i.e., (curried) abstractions remain wrapped up in closures until they have picked up full sets of arguments.

The operations of this SE(M)CD machine may be defined by a state transition function

$$\tau : (S,\ E,\ M,\ C,\ D)\ \{|\ guard\} \rightarrow (S',\ E',\ M',\ C',\ D')\ .$$

It is specified by a finite set of rules that map current into next machine states whose components are the runtime structures involved, with *guard* denoting an optional guard term.

The contents of the stack-like runtime structures are specified as

$$stack =_s nil \mid X \mid item : stack\ ,$$

where $=_s$ again denotes syntactical equality, ' $:$' separates a topmost symbol or expression of interest from the remainder of the structure, and *nil* denotes the empty structure. The symbol X fills in for one of the stack symbols $S,\ E,\ M,\ C,\ D$. Legitimate items in C are variables, constructors and entire λ-expressions. Items in S are values, i.e., variables, applications with something other than abstractions in operator positions, closures and so-called suspensions of the general form $[\ E\ e\ ]$ that pair any unevaluated expression e with an environment E. Suspensions (which include closures) are created for operands that need to be passed on to operators in unevaluated form (or whose evaluation must be suspended), which is what normal order reduction is all about. The environment E contains variable | value pairs of the general form $< v\ ee >$, with ee representing either a variable, a closure or a suspension. Items in M are constructor symbols only, and the machine states stored in the dump D are triples of the form $(E,\ C,\ D)$.

The expressions that this machine is supposed to interpret feature the constructor syntax

$$e\ =_s v \mid \lambda v\ e_b \mid [\ E\ e\ ] \mid \overline{@}\ e_a\ e_f \mid @\ e_f\ e_a\ ,$$

i.e., we have variables v, abstractions (with λv taken as a unary binding constructor and the body e_b as its sole subtree [3]), suspensions (closures), and two

[3] Note that the dot between the constructor λv and the body expression e_b has been dropped to expose both as separate syntactical entities.

types of applications formed by the binary constructors $\overline{@}$ and @. These constructors are also referred to as apply nodes or applicators that respectively are to enforce applicative and normal order reductions. The components e_f and e_a denote expressions in operator (function) and operand (argument) positions, respectively, relative to these apply nodes. The idea is to consider both applications as preorder linearized representations of binary trees, as depicted in Fig. 5.3, where $\overline{@}$ takes its left subexpression as operand and its right subexpression as operator, whereas the apply node @ has both expressions interchanged.

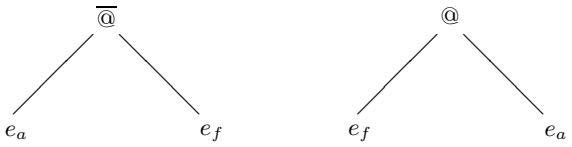


Fig. 5.3. Tree representation of applicative and normal order applications

Scanning the components of the applications $\overline{@}\ e_a\ e_f$ and $@\ e_f\ e_a$ from left to right is equivalent to traversing the respective trees in preorder, i.e., the root node is visited first, followed by a traversal of the left subtree in preorder, followed by a traversal of the right subtree in preorder. This is exactly the order in which we wish to evaluate the components of the applications: the applicator $\overline{@}$ demands that both subexpressions be evaluated in any order (see Sect. 4.5), so we decide to do e_a first and then e_f, whereas the applicator @ demands that only e_f (which is traversed first) be evaluated and that e_a remains as it is.

5.2.1 The Traversal Mechanism

The basic mechanism of the SE(M)CD machine involves the structures C, M and S only. They are operated like a shunting yard to perform preorder traversals of constructor expressions in search of β-redices. To accomplish this, the expressions are initially set up in preorder linearized form in C and moved from there to S. Preorder linearization of the expressions is preserved by temporarily sidelining all constructor symbols – basically the applicators – in M, while their subexpressions are recursively moved from C to S, where they end up in left-right transposed form. In between, there are configurations in which the components of redices are spread out over the tops of the three stacks involved, from where they can be readily removed and replaced by their reductums.

For a formal specification of this preorder traversal by means of state transition rules, and ignoring β-reductions, it suffices to consider only applications

and to treat them summarily as constructor expressions of the general form $ap\ e_0\ e_1$, where $ap \in \{@, \overline{@}\}$, since there is in this case no need to distinguish between the applicative and normal order regimes. Assuming that these applications appear on top of C as $ap : e_0 : e_1 : C$, we have the following traversal rules:[4]

$$\begin{aligned}
(S,\ E,\ M,\ ap : C,\ D) &\ \rightarrow\ (S,\ E,\ ap^2 : M,\ C,\ ;D) \\
(S,\ E,\ ap^i : M,\ e : C,\ D) | (i > 0) &\ \rightarrow\ (e^t : S,\ E,\ ap^{(i-1)} : M,\ C,\ D) \\
(S,\ E,\ ap^0 : nil,\ C,\ D) &\ \rightarrow\ (ap : S,\ E,\ nil,\ C,\ D) \\
(S,\ E,\ ap^0 :\ ap^i : M,\ C,\ D) | (i > 0) &\ \rightarrow\ (ap : S,\ E,\ ap^{(i-1)} : M,\ C,\ D)\ .
\end{aligned}$$

The traversal of a constructor expression begins by moving the apply node from C to M (the first rule). The index $i \in \{0,\ 1,\ 2\}$ attached to the node symbol while it is in M keeps track of the number of its subexpressions that are still in C. This index decrements whenever a complete (sub)expression e has been moved over to S, where it builds up with its left and right subexpressions interchanged, denoted by e^t (the second rule). Whenever the index is down to zero, the apply node itself is moved to S as this completes the traversal from C to S of a (sub)expression of which it is the root node. If there is another apply node underneath it on M, its index is accordingly decremented by one (the last rule). The third rule applies to the special case where, other than for the topmost apply node, M is empty, which terminates the traversal of a complete expression.

Figure. 5.4 illustrates by means of the expression $@\ a\ \overline{@}\ b\ c$ how this traversal works, showing just the contents of the stacks C, S and M. The traversal begins with the stack configuration in the upper left and, following the sequence of arrows, terminates with the configuration in the lower left.

We note that this traversal mechanism brings about a stack configuration (the third configuration in the sequence) in which the applicator @ appears on top of stack M, its left subexpression a – the operator – on top of S, and its right subexpression $\overline{@}\ b\ c$ – the operand – on top of C. Likewise, in the sixth configuration of the sequence, we have the applicator $\overline{@}$ on top of M, and its left subexpression b – the operand – underneath its right subexpression c – the operator – stacked up in S. Thus, we have in S in either configuration the subexpression(s) that need to be evaluated before reducing the respective application. These are both the operand and the operator if the applicator is $\overline{@}$, but just the operator if the applicator is @. If the operator expression happens to be an abstraction, then these configurations may be readily intercepted by the machine to perform (naive) β-reductions. This involves first taking the components of the redex off the stack tops, then creating in the environment E the corresponding variable | value pair, and – after having saved the current machine state on the dump – finally pushing the abstraction body onto C for further evaluation.

[4] For the sake of defining these traversal rules, abstractions $\lambda v\ e_b$ are simply considered expressions e.

$$
\begin{array}{rcl}
@ : a : \overline{@} : b : c : nil \mid C & & a : \overline{@} : b : c : nil \mid C \\
nil \mid M & \Longrightarrow & @^2 : nil \mid M \\
nil \mid S & & nil \mid S \\
 & & \Downarrow \\
b : c : nil \mid C & & \overline{@} : b : c : nil \mid C \\
\overline{@}^2 : @^1 : nil \mid M & \Longleftarrow & @^1 : nil \mid M \\
a : nil \mid S & & a : nil \mid S \\
 & \Downarrow & \\
c : nil \mid C & & nil \mid C \\
\overline{@}^1 : @^1 : nil \mid M & \Longrightarrow & \overline{@}^0 : @^1 : nil \mid M \\
b : a : nil \mid S & & c : b : a : nil \mid S \\
 & & \Downarrow \\
nil \mid C & & nil \mid C \\
nil \mid M & \Longleftarrow & @^0 : nil \mid M \\
@ : \overline{@} : c : b : a : nil \mid S & & \overline{@} : c : b : a : nil \mid S
\end{array}
$$

Fig. 5.4. Traversing the expression $@\ a\ \overline{@}\ b\ c$ from stack C to S via stack M

5.2.2 Doing β-Reductions

The state transition rules necessary to perform β-reductions are given in Fig. 5.5. Together with the rules that govern the traversal of expressions, they completely specify the state transition function τ of the SE(M)CD machine. The rules are listed from top to bottom in the order in which they need to be tried on actual machine states so that the first rule whose left-hand side matches is the one to be executed. Ambiguities may otherwise occur among some of the rules, mainly among those that intercept and execute β-redices and the traversal rules, which are the last ones in the list.

However, we will describe the rules that effect β-reductions in the sequence in which they actually apply to β-redices and their components.

The first rules to be considered are those that create closures for abstractions that appear on top of C. Rule (8) applies to a configuration with an empty stack M, in which case the abstraction is the entire expression to be evaluated. Rule (9) creates closures for abstractions in operator and operand positions relative to either of the apply nodes $\overline{@}$ or $@$ in M. Since closures represent values, they are in both cases set up in stack S.

Applications of closures occurring in operator positions of apply nodes, which in fact constitute instances of β-reduction, are handled by rules (2) and (3). The first of these rules applies to a configuration with an apply

$$(1)\ (S,\ E,\ M,\ nil,\ (E',\ C',\ D')) \rightarrow (S,\ E',\ M,\ C',\ D')$$

$$(2)\ ([\ E'\ \lambda v\ e_b\ ] : {e_a}^t : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \rightarrow (S,\ <v\ e_a>: E', M,\ e_b : nil,\ (E,\ C,\ D))$$

$$(3)\ ([\ E'\ \lambda v\ e_b\ ] : S,\ E,\ @^1 : M,\ e_a : C,\ D) \rightarrow (S,\ <v\ [\ E\ e_a\ ]>: E', M,\ e_b : nil,\ (E,\ C,\ D))$$

$$(4)\ (S,\ E,\ nil,\ v : C,\ D) \rightarrow (lookup(\ v,\ E\ ) : S,\ E,\ nil,\ C,\ D)$$

$$(5)\ (S,\ E,\ ap^i : M,\ v : C,\ D) \mid (i > 0) \rightarrow (lookup(\ v,\ E\ ) : S,\ E,\ ap^{(i-1)} : M,\ C,\ D)$$

$$(6)\ ([\ E'\ e_a\ ] : S,\ E,\ nil,\ C,\ D) \mid (e_a \neq_s \lambda v\ e_b) \rightarrow (S,\ E',\ nil,\ e_a : nil,\ (E,\ C,\ D))$$

$$(7)\ ([\ E'\ e_a\ ] : S,\ E,\ ap^i : M,\ C,\ D) \mid (e_a \neq_s \lambda v\ e_b) \rightarrow (S,\ E',\ ap^{(i+1)} : M,\ e_a : nil,\ (E,\ C,\ D))$$

$$(8)\ (S,\ E,\ nil,\ \lambda v : e_b : C,\ D) \rightarrow ([\ E\ \lambda v\ e_b\ ] : S,\ E,\ nil,\ C,\ D)$$

$$(9)\ (S,\ E,\ ap^i : M,\ \lambda v : e_b : C,\ D) \mid (i > 0) \rightarrow ([\ E\ \lambda v\ e_b\ ] : S,\ E,\ ap^{(i-1)} : M,\ C,\ D)$$

$$(10)\ (S,\ E,\ M,\ ap : C,\ D) \rightarrow (S,\ E,\ ap^2 : M,\ C,\ D)$$

$$(11)\ (S,\ E,\ ap^0 : nil,\ C,\ D) \rightarrow (ap : S,\ E,\ nil,\ C,\ D)$$

$$(12)\ (S,\ E,\ ap^0 : ap^i : M,\ C,\ D) \mid (i > 0) \rightarrow (ap : S,\ E,\ ap^{(i-1)} : M,\ C,\ D)$$

Fig. 5.5. The complete set of state transition rules of the SE(M)CD machine

node $\overline{@}^0$ in M in conjunction with a closure in S and an (evaluated) operand expression underneath. The second rule applies to a configuration with an apply node $@^1$ in M, a closure in S, and an unevaluated operand expression still in E. In either case, the machine sets up as a new environment the one that comes along with the closure, prepends to this environment a new entry that pairs the variable v bound in the abstraction with the value of the operand expression, isolates in C the abstraction body for evaluation in this new environment, and saves in the dump as return continuation the machine state comprising the old environment and the remaining code structure C. The difference between these two rules essentially boils down to the creation of the new environment entry: rule (2) takes the evaluated operand expression off stack S as it is, whereas rule (3) takes the unevaluated operand off the top of C and wraps it up in a suspension $[\ E\ e_a\ ]$ to postpone its evaluation. A suspension looks almost like a closure, except that e_a may be any legitimate λ-expression, not just an abstraction.

The machine then moves on to traverse the abstraction body e_b from C to S. Whenever it encounters a variable v on top of C, it generally applies rule (5) to search, by means of the function *lookup*, the current environment for an entry $< v\ ee >$ and, if successful, returns the value ee on top of S. Otherwise the variable v is free in the entire expression, in which case *lookup* simply returns the variable itself as its own value. Rule (4) takes care of the special case where stack M is empty.

If *lookup* retrieves from E a suspension containing an expression other than an abstraction, it is set up for evaluation in C and E, and the current machine state is saved in D. This action is specified by rule (6) for the general case of a nonempty stack M, and by rule (7) for the special case of an empty stack M.

Upon completing the evaluation of an instantiated abstraction body, the code structure C becomes empty. This machine configuration effects the state transition rule (1). It restores as return continuation the configuration before calling either rule (2) or rule (3) to enter into a β-reduction, but the original redex is replaced with the value found on top of S.

The remaining rules (10) to (12) apply to the handling of apply nodes while traversing an expression from C to S.

Just like the original SECD machine, this machine generally reduces λ-expressions to weak normal forms only, as it neither substitutes nor reduces under abstractors. If the machine starts out with the state $(nil, nil, nil, e : nil, nil)$, where e denotes the expression to be reduced, then it terminates with the state $(\ wnf : nil, nil, nil, nil, nil\ )$ which has the weak normal form wnf of e on S (provided it exists), and with empty structures otherwise. The weak normal form could be a free variable, a closure, or an irreducible application.

Since the machine can perform both applicative and normal order reductions, the applicators should be chosen so as to guarantee termination with a near-minimal number of β-reductions. The choice can be made either by appropriate annotations on parenthesized λ-expressions, or by taking normal order reduction as the default option and determining by some *á* priori analysis which of the applications may safely be reduced in applicative order.

5.2.3 Reducing a Simple Expression

As an example, we consider again the application

$$(((\lambda u.\lambda v.\lambda w.((u\ v)\ w)\ v)\ w)\ u)$$

of Sect. 5.1, whose reduction was illustrated in Figs. 5.1 and 5.2. Since all the arguments of the (nested) application are free variables which, by definition, are their own values, the three β-reductions can safely be done in applicative order. This is generally more efficient since it avoids the overhead of wrapping the arguments up in suspensions whose environments, in this particular case, would be empty.

Now, transforming this λ-expression into constructor syntax, with $\overline{@}$ as the apply node, means that we need to replace systematically parenthesized expressions of the form $(e_0\ e_1)$ by $\overline{@}\ e_1\ e_0$, which gives[5]

$$\overline{@}\ u\ \overline{@}\ w\ \overline{@}\ v\ \lambda u\ \lambda v\ \lambda w\ \overline{@}\ w\ \overline{@}\ v\ u$$

Figure. 5.6 shows how the SE(M)CD machine reduces this expression.

The initial machine state (or stack configuration) has the expression set up in the code structure C; all other structures are empty.[6] The machine then traverses the expression from C to S and tries to evaluate its components. Since the arguments $v,\ w,\ u$ are already values, the only thing left to do before reaching the second configuration from the top is to create for the abstraction a closure with an empty environment.

With this closure on top of S, the argument v underneath and an applicator $\overline{@}^0$ on top of M, the machine recognizes a redex to which rule (2) applies. It pops the applicator off M, removes the closure from S and unwraps it, takes the abstractor λu off the abstraction and removes the argument v from S, creates from these two items an entry $< u\ v >$ that it adds to the environment E' carried along with the closure. It also saves the structures E, C and recursively D (all three of which are as yet empty) in the dump D, and sets the remaining abstraction up for evaluation in C, as shown in the third configuration from the top.

Creating a closure in S and applying it to an argument is repeated twice to reduce the remaining two redices. This results in the fourth configuration from the top, which has the abstraction body set up in C and the environment in which it is to be evaluated set up in E. While traversing the abstraction body from C to S, every variable that pops to the top of C is replaced by its associated value found in the environment, which is pushed onto S, resulting in the second to last configuration shown in the figure.

With the reduced expression in left-right transposed form in S, the machine has basically done its job, except that the current environment is still in E, and the dump D still contains nestings of machine configurations that include the previous states of the environment. Both structures are cleaned up by repeated application of the first state transition rule so that the final state has all structures other than S empty.

Knowing that the expression in S is a preorder linearized representation of a tree that has its left and right subtrees flipped, we obviously have, relative to an applicator $\overline{@}$, the operator in the left subtree and the operand in the right subtree. This means that we can convert this expression back into an

[5] It does not really matter which applicators are used in the abstraction body since the variables are substituted by other variables, so the applications will not become redices.

[6] Note that all occurrences of constructor symbols and variables in the C, M and S structures are now separated by ':', and so are the variable | value entries in the environment E.

$$\begin{array}{r} nil \mid S \\ nil \mid E \\ nil \mid M \\ \overline{@} : u : \overline{@} : w : \overline{@} : v : \lambda u : \lambda v : \lambda w : \overline{@} : w : \overline{@} : v : u : nil \mid C \\ nil \mid D \end{array}$$

repeatedly rules (10) and (5), and then rule (9) $\Downarrow$

$$\begin{array}{r} [\ nil\ \lambda u\ \lambda v\ \lambda w\ \overline{@}\ w\ \overline{@}\ v\ u\ ] : v : w : u : nil \mid S \\ nil \mid E \\ \overline{@}^0 : \overline{@}^1 : \overline{@}^1 : nil \mid M \\ nil \mid C \\ nil \mid D \end{array}$$

rule (2) $\Downarrow$

$$\begin{array}{r} w : u : nil \mid S \\ < u\ v > : nil \mid E \\ \overline{@}^1 : \overline{@}^1 : nil \mid M \\ \lambda v : \lambda w : \overline{@} : w : \overline{@} : v : u : nil \mid C \\ (nil,\ nil,\ nil) \mid D \end{array}$$

rules (9), (2) twice $\Downarrow$

$$\begin{array}{r} nil \mid S \\ < w\ u > :< v\ w > :< u\ v > : nil \mid E \\ nil \mid M \\ \overline{@} : w : \overline{@} : v : u : nil \mid C \\ (< v\ w > :< u\ v > : nil,\ nil, (< u\ v > : nil,\ nil,\ (nil,\ nil,\ nil))) \mid D \end{array}$$

repeatedly traversal rules (10) to (12) and rule (5) $\Downarrow$

$$\begin{array}{r} \overline{@} : \overline{@} : v : w : u : nil \mid S \\ < w\ u > :< v\ w > :< u\ v > : nil \mid E \\ nil \mid M \\ nil \mid C \\ (< v\ w > :< u\ v > : nil,\ nil, (< u\ v > : nil,\ nil,\ (nil,\ nil,\ nil))) \mid D \end{array}$$

rule (1) three times $\Downarrow$

$$\begin{array}{r} \overline{@} : \overline{@} : v : w : u : nil \mid S \\ nil \mid E \\ nil \mid M \\ nil \mid C \\ nil \mid D \end{array}$$

Fig. 5.6. Reducing $(((\lambda u.\lambda v.\lambda w.((u\ v)\ w)\ v)\ w)\ u)$ applicative order with the SE(M)CD machine

equivalent parenthesized form as

$$\overline{@}\ \overline{@}\ v\ w\ u\ \rightarrow\ ((v\ w)\ u)\ \ ,$$

i.e., the result – as expected – is the same as the one obtained by the reduction sequences in Figs. 5.1 and 5.2.

5.3 The #SE(M)CD Machine for the Nameless Λ-Calculus

It takes only a few minor modifications of the SE(M)CD machine to make it reduce expressions of the nameless Λ-calculus introduced in Sect. 4.4, which uses binding indices instead of variables as placeholders for substitutions.

The main advantage of switching to the Λ-calculus relates to the representation of and accesses to the environment. We have seen in the preceding section that the SE(M)CD machine, when it reduces the application

$$(((\lambda u.\lambda v.\lambda w.((u\ v)\ w)\ v)\ w)\ u)\ \ ,$$

constructs an environment of variable | value pairs

$$< w\ u >:< v\ w >:< u\ v >: nil$$

for the evaluation of the abstraction body $((u\ v)\ w)$, i.e., the entry for the innermost bound variable w ends up at the top of stack E and the entry for the outermost bound variable u ends up at the bottom (compare Fig. 5.6).

If we now α-convert this application into the nameless Λ-calculus, we obtain

$$(((\Lambda.\Lambda.\Lambda.((\#2\ \#1)\ \#0)\ v)\ w)\ u)\ \ .$$

An environment for the evaluation of the abstraction body $((\#2\ \#1)\ \#0)$ that would be created the same way the SE(M)CD does it but uses binding indices instead of variables would obviously have to look like this:

$$< \#0\ u >:< \#1\ w >:< \#2\ v >: nil\ \ .$$

We immediately recognize that the binding indices that have replaced the variables identify directly the positions of the entries relative to the top of E. We can therefore safely drop these indices altogether and just take the values associated with them as environment entries.

The state transition rules of this modified #SE(M)CD machine are given in Fig. 5.7. They are the same as those of the SE(M)CD machine, except that

- λ-abstractors are now replaced by Λ-abstractors;
- binding indices replace occurrences of bound variables;
- expressions represented in this form are denoted as $\#e$;
- entries in the environment are just values (including suspensions);

$$
\begin{array}{ll}
(1) & (S,\ E,\ M,\ nil,\ (E',\ C',\ D')) \rightarrow (S,\ E',\ M,\ C',\ D') \\
\\
(2) & ([\ E'\ \Lambda\ \#e_b\ ] : \#{e_a}^t : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \\
 & \qquad \rightarrow (S,\ \#e_a : E', M,\ \#e_b : nil,\ (E,\ C,\ D)) \\
(3) & ([\ E'\ \Lambda\ \#e_b\ ] : S,\ E,\ @^1 : M,\ \#e_a : C,\ D) \\
 & \qquad \rightarrow (S,\ [\ E\ \#e_a\ ] : E', M,\ \#e_b : nil,\ (E,\ C,\ D)) \\
\\
(4) & (S,\ E,\ nil,\ \#j : C,\ D) \\
 & \qquad \rightarrow\ (lookup(\ \#j,\ E\ ) : S,\ E,\ nil,\ C,\ D) \\
(5) & (S,\ E,\ ap^i : M,\ \#j : C,\ D) \mid (i > 0) \\
 & \qquad \rightarrow (lookup(\ \#j,\ E\ ) : S,\ E,\ ap^{(i-1)} : M,\ C,\ D) \\
\\
(6) & ([\ E'\ \#e_a\ ] : S,\ E,\ nil,\ C,\ D) \mid (\#e_a \neq_s \Lambda\ \#e_b) \\
 & \qquad \rightarrow (S,\ E',\ nil,\ \#e_a : nil,\ (E,\ C,\ D)) \\
(7) & ([\ E'\ \#e_a\ ] : S,\ E,\ ap^i : M,\ C,\ D) \mid (\#e_a \neq_s \Lambda\ \#e_b) \\
 & \qquad \rightarrow (S,\ E',\ ap^{(i+1)} : M,\ \#e_a : nil,\ (E,\ C,\ D)) \\
\\
(8) & (S,\ E,\ nil,\ \Lambda : \#e_b : C,\ D) \\
 & \qquad \rightarrow ([\ E\ \Lambda\ \#e_b\ ] : S,\ E,\ nil,\ C,\ D) \\
(9) & (S,\ E,\ ap^i : M,\ \Lambda : \#e_b : C,\ D) \mid (i > 0) \\
 & \qquad \rightarrow ([\ E\ \Lambda\ \#e_b\ ] : S,\ E,\ ap^{(i-1)} : M,\ C,\ D) \\
\\
(10) & (S,\ E,\ M,\ ap : C,\ D) \rightarrow\ (S,\ E,\ ap^2 : M,\ C,\ D) \\
(11) & (S,\ E,\ ap^0 : nil,\ C,\ D) \rightarrow\ (ap : S,\ E,\ nil,\ C,\ D) \\
(12) & (S,\ E,\ ap^0 : ap^i : M,\ C,\ D) | (i > 0) \\
 & \qquad \rightarrow\ (ap : S,\ E,\ ap^{(i-1)} : M,\ C,\ D)
\end{array}
$$

Fig. 5.7. The state transition rules for the #SE(M)CD machine

- the function *lookup* accesses the actual environment using the binding indices as offsets relative to the topmost entry.

Figure 5.8 shows the state transitions effected by these rules on the above Λ-expression. Other than for the syntactical changes and the representation of the environment entries, they are exactly the same as in Fig. 5.6.

5.4 Implementing δ-Reductions

Both the SE(M)CD machine and the #SE(M)CD machine may be extended without problems to support an applied λ-calculus as well.

To get the basic idea across, it suffices just to give the state transition rules for addition and 'greater than' comparison. Apart from the specifics of the particular functions, all rules for arithmetic, logic and relational operations basically look the same. As a matter of convenience, we will also assume that none of the applications produces type inconsistencies that would require special treatment, as outlined in Sects. 4.7 and 4.8.

$$\begin{array}{r} nil \mid S \\ nil \mid E \\ nil \mid M \\ \overline{@} : u : \overline{@} : w : \overline{@} : v : \Lambda : \Lambda : \Lambda : \overline{@} : \#0 : \overline{@} : \#1 : \#2 : nil \mid C \\ nil \mid D \end{array}$$

repeatedly rules (10) and (5), and then rule (9) $\Downarrow$

$$\begin{array}{r} [\ nil\ \Lambda\ \Lambda\ \Lambda\ \overline{@}\ \#0\ \overline{@}\ \#1\ \#2\ ] : v : w : u : nil \mid S \\ nil \mid E \\ \overline{@}^0 : \overline{@}^1 : \overline{@}^1 : nil \mid M \\ nil \mid C \\ nil \mid D \end{array}$$

rule (2) $\Downarrow$

$$\begin{array}{r} w : u : nil \mid S \\ v : nil \mid E \\ \overline{@}^1 : \overline{@}^1 : nil \mid M \\ \Lambda : \Lambda : \overline{@} : \#0 : \overline{@} : \#1 : \#2 : nil \mid C \\ (nil,\ nil,\ nil) \mid D \end{array}$$

rules (9), (2) twice $\Downarrow$

$$\begin{array}{r} nil \mid S \\ u : w : v : nil \mid E \\ nil \mid M \\ \overline{@} : \#0 : \overline{@} : \#1 : \#2 : nil \mid C \\ (w : v : nil,\ nil, (v : nil,\ nil,\ (nil,\ nil,\ nil))) \mid D \end{array}$$

repeatedly rules (10) to (12) and rule (5) $\Downarrow$

$$\begin{array}{r} \overline{@} : \overline{@} : v : w : u : nil \mid S \\ u : w : v : nil \mid E \\ nil \mid M \\ nil \mid C \\ (w : v : nil,\ nil, (v : nil,\ nil,\ (nil,\ nil,\ nil))) \mid D \end{array}$$

rule (1) three times $\Downarrow$

$$\begin{array}{r} \overline{@} : \overline{@} : v : w : u : nil \mid S \\ nil \mid E \\ nil \mid M \\ nil \mid C \\ nil \mid D \end{array}$$

Fig. 5.8. Reducing $(((\Lambda.\Lambda.\Lambda.((\#2\ \#1)\ \#0)\ v)\ w)\ u)$ in applicative order with the #SE(M)CD machine

To comply with the syntax supported by either machine, binary arithmetic operations need to be represented as twofold nestings of unary applications. Since arithmetic operations require that their operands be evaluated before the operators can be applied, such applications need to be expressed in constructor syntax as

$$\overline{@}\, e_2\, \overline{@}\, e_1\, \mathit{arith\_op}$$

for applicative order evaluation.

Assuming that e_1 and e_2 are (or evaluate to) numbers num_1 and num_2, respectively, and that the operator is $+$, this application is supposed to reduce in two steps as follows:

$$\overline{@}\, num_2\, \overline{@}\, num_1\, + \;\rightarrow_\delta\; \overline{@}\, num_2\, \{+\, num_1\} \;\rightarrow_\delta\; (num_2 + num_1)\;\;,$$

where $\{+\, num_1\}$ denotes an intermediary function that reads 'add num_1 to whatever argument', and $(num_2 + num_1)$ represents the sum of num_1 and num_2. The SE(M)CD machine can do these δ-reductions using the rules

$$(+ : num_1 : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \rightarrow (S,\ E,\ M,\ \{+\, num_1\} : C,\ D)$$

and

$$(\{+\, num_1\} : num_2 : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \rightarrow (S,\ E,\ M,\ (num_2 + num_1) : C,\ D)$$

Likewise, binary relational operations need to be represented in constructor syntax as $\overline{@}\, e_2\, \overline{@}\, e_1\, \mathit{rel\_op}$. An application $\overline{@}\, num_2\, \overline{@}\, num_1\, \mathtt{gt}$ reduces, again in two steps, as:

$$\overline{@}\, num_2\, \overline{@}\, num_1\, \mathtt{gt} \rightarrow_\delta \overline{@}\, num_2\, \{\, \mathtt{gt}\, num_1\, \} \rightarrow_\delta \begin{cases} \lambda s \lambda t\ s\ \mathtt{if}\ (num_2 > num_1) \\ \\ \lambda s \lambda t\ t\ \mathtt{if}\ (num_2 \leq num_1) \end{cases}$$

(here we need to remember from Sect. 4.7 that the combinators $\lambda s \lambda t\ s$ and $\lambda s \lambda t\ t$ stand for the Boolean values `true` and `false`, respectively, that, in the larger context of `if_then_else` clauses, select between consequent and alternative). This translates into the SE(M)CD machine rules[7]

$$(\mathtt{gt} : num_1 : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \rightarrow (S,\ E,\ M,\ \{\mathtt{gt}\, num_1\} : C,\ D)$$

and

[7] Two more rules of the same kind take care of arguments that are both character strings.

$$(\{\texttt{gt}\ num_1\} : num_2 : S,\ E,\ \overline{@}^0 : M,\ C,\ D) \rightarrow \\ \texttt{if}\ (num_2 > num_1) \\ \texttt{then}\ (S, E,\ M,\ \lambda s\ \lambda t\ s : C,\ D) \\ \texttt{else}\ (S, E,\ M,\ \lambda s\ \lambda t\ t : C,\ D)\ .$$

Another rule is necessary to deal with the special case where stack M is empty and a δ-reduction rule has left a value *val* (which may also be some intermediary function) in C. This rule simply moves this value from C to S:

$$(S,\ E,\ nil,\ val : C,\ D) \rightarrow (val : S,\ E,\ nil,\ C,\ D)$$

Very similar rules may be defined for δ-reductions involving other arithmetic, logic and relational primitives, and also for the list processing functions discussed in Chap. 3 and Sect. 4.7.

5.5 Other Weakly Normalizing Abstract Machines

There are two more abstract machines that ought to be mentioned for reasons of completeness as they (or their descendants) also play a major role in implementing a weakly normalizing λ-calculus. They share with the SE(M)CD machines the notions of delayed substitutions, environments and closures or suspensions.

5.5.1 The $\mathcal{K}$-Machine

Krivine's extremly simple $\mathcal{K}$-machine supports a normal order regime for the nameless Λ-calculus. It uses just two structures T and E which respectively hold the Λ-expressions to be transformed and an environment whose entries are suspensions. A third structure S serves as a working stack for the temporary storage of suspensions. The T-structure in fact replaces the combined structures C, M and S of the SE(M)CD machines as far as the traversal of expressions in search of redices is concerned. This higher level of abstractions renders the specification of the machine more concise and keeps more options open for an implementation.

The syntax of the Λ-expressions processed by this machine is

$$e =_s \#i \mid (e_0\ e_1) \mid \Lambda e \mid [\ E\ e\ ]\ ,$$

and its state transition function is defined as

$$\tau_{\mathcal{K}} : (E,\ T,\ S) \rightarrow (E',\ T',\ S')\ .$$

This function is realized by just four rules, corresponding to the three syntactical figures (with an extra rule for the binding index #0), as:

$$
\begin{aligned}
([\ E_1\ e_1\ ] : E,\ \#0,\ S) &\rightarrow (E_1,\ e_1,\ S) \\
(v : E,\ \#(i+1),\ S) &\rightarrow (E,\ \#i,\ S) \\
(E,\ (e_0\ e_1),\ S) &\rightarrow (E,\ e_0,\ [\ E\ e_1\ ] : S) \\
(E,\ (\Lambda e_0\ ),\ s : S) &\rightarrow (s : E,\ e_0,\ S)\ .
\end{aligned}
$$

The first and the second rule combined select from the environment the i-th suspension relative to the top, thus replacing the function *lookup* of the SE(M)CD machine, the third rule creates a suspension in S for the operand of an application, and the fourth rule has, under the control of an abstraction in T, a suspension removed from the top of S and pushed as an entry into E, thereby also exposing the body of the abstraction in T.

The initial machine configuration has the expression set up in T; both the environment and the stack are empty. If everything goes right, the machine terminates for all closed Λ-expressions with a weak normal form in T, and with E and S again empty.

Figure. 5.9 shows, as a sequence of state transformations, how the $\mathcal{K}$-machine reduces the Λ-expression $(\Lambda(\#0\ \#0)\ \Lambda\#0)$ to $\Lambda\#0$.

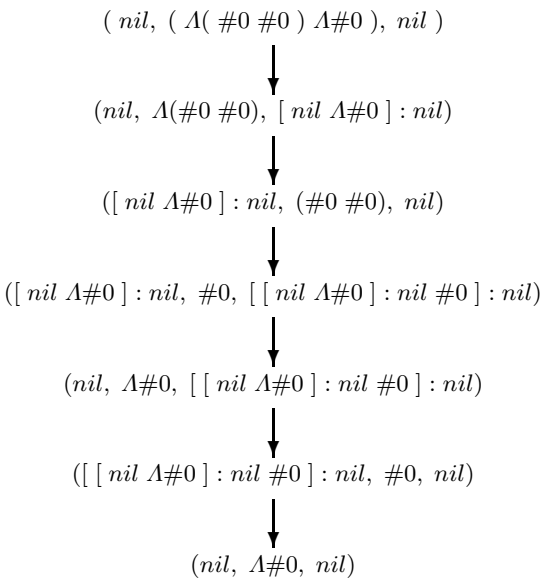


Fig. 5.9. Reducing a Λ-expression in the $\mathcal{K}$-machine

There exist numerous variants and extensions of this machine, of which one will be described in the next chapter. It is also the subject of extensive theoretical work.

5.5.2 The Categorial Abstract Machine

The other machine at which we should have a brief look is the categorial abstract machine (CAM) of Cousineau, Curien and Mauny which implements an applicative order regime for the Λ-calculus. The attribute 'categorial' essentially relates to the notion of products (or pairs) which are used to construct (nested) terms of the form $< p,\ q >$ that represent environments composed of closures. The operations on these terms include selection of the first or second component, swapping, and *cons*ing (composing a term from two subterms) that suggest realization as instructions.

The basic idea of the CAM therefore is to translate expressions of the pure Λ-calculus into code which, in addition to the instructions just mentioned, includes others that deal with abstractions and applications, and to have this code construct and operate on terms.

To do so, the CAM supports three runtime structures, of which T and C respectively hold the terms and the code, and S again serves as a working stack.[8] The state transition function of the CAM is defined as

$$\tau_{cam} : (T,\ instr : C,\ S) \to (T',\ C',\ S')$$

(with *instr* denoting any of the instructions which, upon execution, is removed from the code structure).

The state transition rules for the CAM instructions are the following:

$$\begin{aligned}
(< p,\ q >,\ \texttt{car} : C,\ S) &\to (p,\ C,\ S)\\
(< p,\ q >,\ \texttt{cdr} : C,\ S) &\to (q,\ C,\ S)\\
(p,\ (\ \texttt{cur}\ c) : C,\ S) &\to ([\ p\ c\ ],\ C,\ S)\\
(p,\ \texttt{push} : C,\ S) &\to (p,\ C,\ p : S)\\
(p,\ \texttt{swap} : C,\ q : S) &\to (q,\ C,\ p : S)\\
(p,\ \texttt{cons} : C,\ q : S) &\to (< q\ p >,\ C,\ p : S)\\
(< [\ p\ c\ ]\ q >,\ \texttt{app} : C,\ S) &\to (< p\ q >,\ c : C,\ S)\ .
\end{aligned}$$

The instructions `car` and `cdr` just select the first and second component of the term in T (first and second rules from the top), the instruction `cur` creates in T a closure for the abstraction code c in the environment p held in T (third rule), and `push` copies the term (environment) p from T to S (fourth rule). The instruction `swap` exchanges the terms p in T and q in S, and `cons` combines them into a new term in T (fifth and sixth rule). Finally, `apps` evaluates a closure found in the first component of the term in T (last rule).

Translating the three syntactical figures of the Λ-calculus into code composed of these instructions is accomplished by the following rules:

[8] Note that the term structure T and the code structure C of the CAM correspond to the environment E and to the term structure T, respectively, of the $\mathcal{K}$-machine, i.e., the terms have different meanings in both machines.

$$
\begin{aligned}
code[\ \#0\ ] &= \texttt{cdr} \\
code[\ \#(i+1)\ ] &= \texttt{car} : code[\ \#i\ ] \\
code[\ \Lambda e\ ] &= (\ \texttt{cur}\ code[\ e\ ]\ ) \\
code[\ (e_0\ e_1)\ ] &= \texttt{push} : code[\ e_0\ ] : \texttt{swap} : code[\ e_1\ ] : \texttt{cons} : \texttt{app}\ .
\end{aligned}
$$

Translating binding indices #i into cars and cdrs suggests construction of the environment by the recursive nesting of terms along the first (head) components of the binary lists, the instruction cur that precedes abstraction code effects currying, and the code for applications is to construct a pair $< code[\ e_0\ ]\ ;\ code[\ e_1\ ] >$, with $<$, ; and $>$ translating into push, swap and cons, respectively. The instruction app simply applies the first (operator) component to the second (operand) component of this pair.

Reducing a Λ-expression e begins with an empty pair in T, an empty stack S, and the code of e in C, i.e., with a configuration $(<>,\ code[\ e\ ] : nil,\ nil)$. It terminates with a term in T representing the weak normal form of e, and with an empty code structure and an empty stack.

To see how this works, we consider again the expression $(\Lambda\ (\#0\ \#0)\ \Lambda\ \#0)$ that, after two β-reductions, returns $\Lambda\ \#0$. Applying to the initial expression the above translation rules yields the code:

$$\texttt{push} : (\texttt{cur}\ \ f) : \texttt{swap} : (\texttt{cur}\ \ \texttt{cdr}) : \texttt{cons} : \texttt{app}\ ,$$

where $f\ =_s\ \texttt{push} : \texttt{cdr} : \texttt{swap} : \texttt{cdr} : \texttt{cons} : \texttt{app}$.

Figure 5.10 shows, as a sequence of state transitions, how this code executes. The computation stops after 13 steps with a configuration that has both the code structure and the stack empty, i.e., it has arrived at a legitimate terminal state. The resulting term in T is a closure containing an empty environment and code that consists of just the instruction cdr. We can convince ourselves that this is the CAM term representing the abstraction $\Lambda\ \#0$ by the following consideration.

We know that an abstraction compiles to curried code which creates a new closure in T that takes the current term in T as the environment. That is to say, an n-ary abstraction would create in T an n-fold nesting of closures. The flat closure that we get in T as the result of our example program thus tells us that we have an abstraction of arity one. Since the instruction cdr that makes up the code of the abstraction body retranslates straightforwardly into the binding index #0, the weak normal form represented by this closure is indeed $\Lambda\ \#0$, as expected.

5.6 Summary

This chapter has focused on some very basic weakly normalizing abstract machines for the pure λ-calculus that play a major role in language implementations, specifically in the functional domain, but also in numerous theoretical studies. Of primary interest are two variants of Landin's SECD machine that

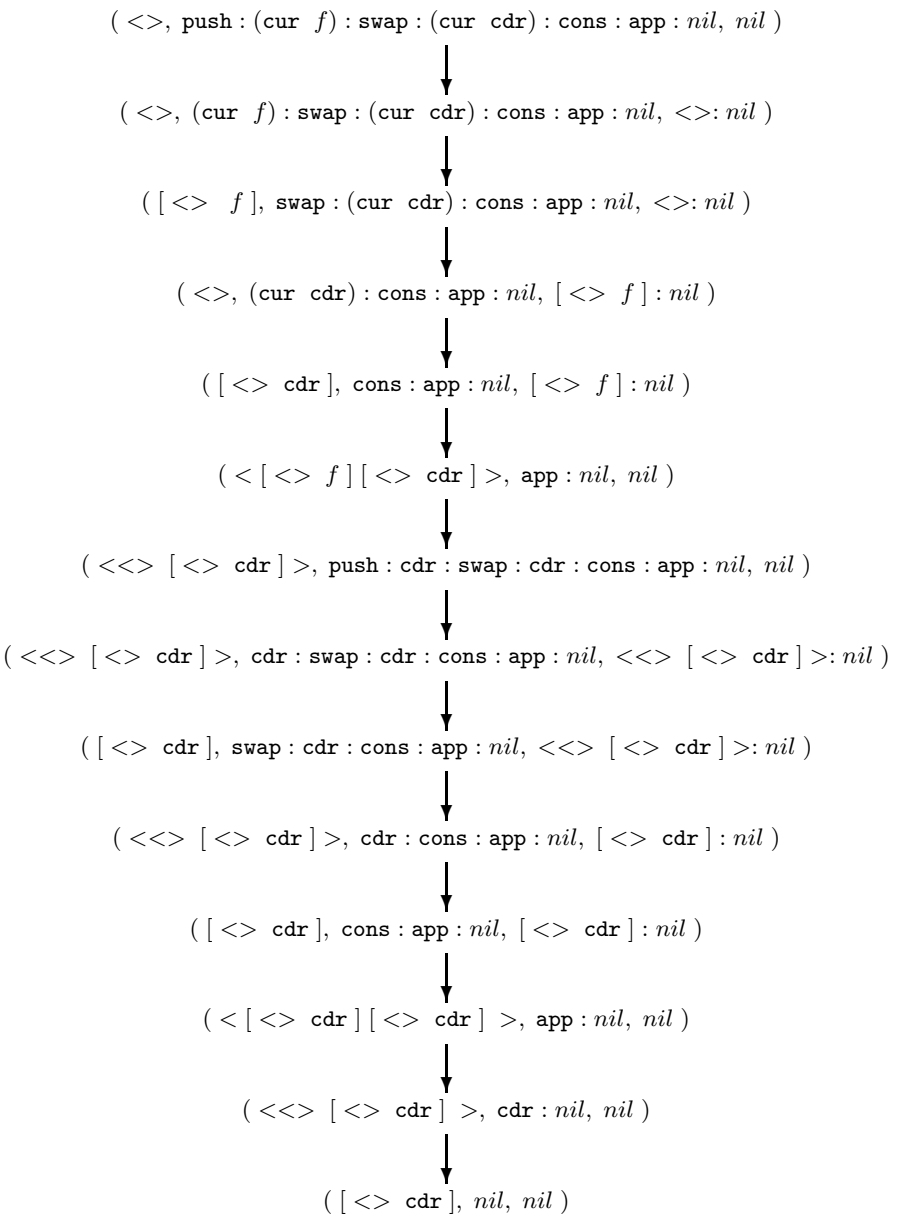


Fig. 5.10. Code execution on the categorial abstract machine

support both applicative and normal order reductions. To do so, the machine represents λ-expressions in a constructor syntax that uses two apply nodes (or applicators) $\overline{@}$ and @ to distinguish between both evaluation strategies.

The most important idea of these SE(M)CD machines is that of delayed substitutions. β-reductions are split up into the collection of variable | value pairs in a runtime structure called the environment and the actual substitutions of bound-variable occurrences that are all done at once when the respective body expressions are subsequently traversed. Closely related to delayed substitutions and environments are closures and suspensions that respectively pair abstractions and, more generally, expressions with the environments in which they may have to be evaluated later on. These features combined are the keys to efficient language implementations.

The basic mechanism of the SE(M)CD machine is a preorder traversal of constructor expressions, with the code structure C as the source of unevaluated expressions, S as the sink stack for expression values and M as an intermediate stack for apply nodes. This traversal brings about stack configurations in which the components of redices are spread out over the tops of the stacks C, M and S, ready to be intercepted and replaced by their reductums.

One of the SE(M)CD machines implements the λ-calculus of named variables, the other implements the nameless Λ-calculus. Except for differences relating to the representation of the environments and to the syntax, the state transition rules of both machines are exactly the same.

δ-reductions are fairly simple and straightforward to implement in both machines. They affect just the contents of the stacks S and M, but neither the environment E nor the dump D. This tells us that primitive operations are obviously not a primary concern when it comes to designing abstract machines. What really matters, and makes things complicated, is the machinery that supports function calls and correctly handles the scoping of variables or of binding indices.

Delayed substitutions, environments and closures (or suspensions) are the concepts shared by the other two machines briefly discussed in this chapter, namely Krivine's $\mathcal{K}$-machine and the categorial abstract machine (CAM) of Cousineau, Curien and Mauny. The former is an extremly simple direct interpreter of Λ-expressions that works with only four state transition rules to transform expressions, using just an environment and a stack as runtime structures. The latter converts Λ-expressions into simple code that basically operates on a term structure and a stack to produce terms as representations of weak normal forms.

It should be clearly understood that the SE(M)CD machines, the $\mathcal{K}$-machine and the CAM just specify what needs to be done in principle, and specifically what runtime structures need to be supported, to weakly normalize λ-expressions, which can be accomplished without engaging in full-fledged β-reductions.

A direct implementation of the SE(M)CD machines and of the $\mathcal{K}$-machine would perform rather poorly as they are based on what is commonly known as string reduction. They interpret linear representations of expressions, and in doing so, involve a lot of copying, of forming and undoing closures (or suspensions), and of moving (copying) entire environments structures and

code fragments from one place to another, e.g., to and from the dump of the SE(M)CD machine. The construction of this dump itself is highly redundant as it may contain repeatedly the same substructures. All this adds up to runtime complexities of order $O(n^2)$ for problems whose representations are of order $O(n)$. The CAM is doing a lot better in this respect as it executes compiled code to construct and operate on terms, primarily closures.

References

The original SECD machine was published by Landin in [Lan64] as a strict evaluator for λ-expressions. Descriptions of this machine may also be found in other textbooks on functional programming and its implementation, e.g., in [Kog91, HaMi99]. The modifications that resulted in the SE(M)CD machines primarily concern the shunting-yard mechanism for traversing constructor expressions, which was first proposed by Berkling for his λ-calculus machine and published in [Ber75]. Doing both applicative and normal order reductions also derives from Berkling's machine, and so does the implementation of the nameless λ-calculus, of which a full account is given in an internal report [Ber76].

The categorial abstract machine of Cousineau, Curien and Mauny has been published in various places, for instance in [CCM85/87]. Interestingly enough, there exists no original publication for Krivine's $\mathcal{K}$-machine [Kri85], but definitions may, for instance, be found in [Cre90, FGSW03].

6

Toward Full-Fledged λ-Calculus Machines

The abstract machines described in the preceding chapter realize a weakly normalizing λ-calculus. β-reductions are performed naively, the risk of potential name clashes is consequently avoided by outlawing substitutions and reductions under λ-abstractors. The values of abstractions are represented as closures, i.e., by constructs that in fact leave abstractions unevaluated.

We could be content with this situation, arguing that this is just a minor inconvenience of not much practical relevance, particularly if we are interested in computing only basic values (or ground terms), and therefore not worth the effort of implementing a supposedly rather complex and time-consuming β-reduction. In fact, all conventional programming languages and computing systems are based on the naive substitution of function parameters. Moreover, they demand that functions (or procedures) be applied to full sets of arguments. Most importantly, variables just represent values but are not values themselves. So, there can be no free variables and hence there can be no name clashes that need to be resolved.

However, we could also argue that too much is given up too easily. Full normalization based on full-fledged β-reductions is an essential prerequisite for symbolic computations involving free variables. It is the key to correctly treating both functions (abstractions) and variables truly as first-class objects, just like values, meaning that they may legitimately occur in both operator and operand positions of applications, and also be returned as function values.

Full normalization is, for instance, required in proof assistant systems, and more specifically in proof checkers based on type theories, where the terms of so-called dependent types must be compared for β-equivalence. In theorem proving (which is the ultimate goal of proof assistant systems) full normalization is one of the more important proof tactics to establish reflexive equality between two terms (see appendix B for an example).

Another area of applying full normalization to advantage are high-level program optimizations such as converting partial applications of abstractions into new, specialized abstractions, normalizing abstraction bodies, or unrolling recursive abstractions to eliminate repeated parameter passing. Such on-the-

fly simplifications could pay off significantly in terms of runtime efficiency if the abstractions are repeatedly applied in different contexts.

Clearly, these optimizations cannot be done with the SE(M)CD machines as they are (and neither with the $\mathcal{K}$-machine nor with the CAM) since they in fact treat closures rather than abstractions as first class objects, with the consequence that instead of symbolically simplifying abstraction bodies only once, the equivalent computational steps may have to be repeated as many times as the closures (rather than the normalized abstractions) are applied elsewhere.

As an example, we consider again the function *double_twice* introduced in Sect. 2.1. We remember that it may be computed by applying the function $twice =_s \lambda f.\lambda v.(f\ (f\ v))$ to itself:

$$(\lambda f.\lambda v.(f\ (f\ v))\ \lambda f.\lambda v.(f\ (f\ v)))\ .$$

It takes five β-reductions as defined in Sect. 4.3, i.e., using protection keys to resolve naming conflicts, to reduce this partial application to

$$double\_twice =_s \lambda v.\lambda v.(/v\ (/v\ (/v\ (/v\ v))))\ .$$

Here the abstraction body has been evaluated as far as is possible without actually applying the abstraction to arguments.

However, the SE(M)CD machine would wrap this partial application up in a closure[1]

$$[\ <f\ \lambda f.\lambda v.(f\ (f\ v))>\ \lambda v.(f\ (f\ v))\ ]$$

and pass it along until it can be applied to a second argument, hopefully to another abstraction. This application would prepend to the environment another entry for the variable v, after which the closure could be unwrapped again and evaluated. Substituting in the abstraction body then exposed occurrences of the variable f by the entry found for it in the environment, and performing subsequent reductions, is equivalent to directly β-reducing the partial application of *twice* to itself. The problem is that these substitutions need to be repeated wherever the closure is applied, rather than doing the equivalent β-reductions only once to compute a fully normalized (or optimized) version of the self-application first.

Beyond avoiding such redundancies, efficiency of execution also relates to the question of how many β-reductions it takes to fully normalize a given λ-expression. Minimizing the length of a reduction sequence, which may be a primary concern, is a matter of choosing the right redex in a particular state of the computation. However, optimizing strategies to this effect are for all practical purposes unattainable for the simple reason that they inflict a considerable runtime overhead, owing to elaborate on-the-fly decision making

[1] The environment is denoted as a list of entries that are of the general form $<var\ e>$.

as to which redex needs to be contracted next. This overhead tends to neutralize any gains made by saving a few β-reductions. Simple strategies such as normal order may duplicate work by reducing operand expressions in possibly several places of substitution, whereas the applicative order regime trades the efficiency of reducing an operand expression exactly once for some risk of getting trapped in runaway recursions even if normal forms exist.

In this chapter, we will give an overview of early implementation techniques for full-fledged β-reductions and also of more recent concepts that successfully combine efficient environment-based β-reductions with a strategy that represents a good compromise between the efficiency of the applicative order regime and the full normalization property of the normal order regime. In subsequent chapters we will develop abstract machines that implement these concepts.

6.1 Berkling's String Reduction Machine

Inspired by John Backus's earlier work on reduction languages, Berkling came up as early as 1975 with a novel way of architecting computing machines that directly implements an applied λ-calculus as the machine language, with a full-fledged β-reduction as its most important operation, and with support for both applicative and normal order evaluation.

This machine was designed with hardware implementation in mind, providing a high-level interface for the λ-calculus. It uses the same linear representation of λ-expressions and the same stack-based shunting yard mechanism for traversing them in search of reducible applications as do the SE(M)CD machines of the preceding chapter.[2]

The constructor syntax of the expressions that can be directly interpreted by the machine is given as

$$e =_s v \mid c \mid pf \mid \lambda u\ e_b \mid \overline{@}\ e_a\ e_f \mid @\ e_f\ e_a \mid <\ e_h\ e_t\ ,$$

i.e., expressions are variables, constant values, primitive functions, abstractions, applicative and normal order applications, again with e_f and e_a respectively denoting operator (function) and operand (argument) expressions, and binary lists. The abstractors λu and the applicators $\overline{@}$, @ are again constructors, and so is the list symbol $<$ that combines a head expression e_h with a tail expression e_t (which may be an end-of-list symbol $>$).

The variables may be preceded by one or several protection keys (or unbinding operators /), i.e., their syntax should be defined more precisely as $v =_s z \mid /v$ (with z being a variable as well). The positions of operators and operands relative to the applicators $\overline{@}$ and @ are, as in the SE(M)CD machines, again chosen in compliance with the order in which they are visited by the underlying preorder traversal mechanism.

[2] In fact, this traversal mechanism was first implemented in Berkling's machine and in this text adopted for the specification of the SE(M)CD machines.

The machine performs β-reductions strictly and directly by the rules for the manipulation of protection keys for bound variable occurrences as given in Sect. 4.3. Rather than using an environment to delay substitutions, as the SE(M)CD machines do, β-reduction is carried out as an atomic operation in the sense that the abstraction body is traversed immediately following the detection of the β-redex to substitute at once all free occurrences of the bound variable in question, and without any other activity in between. Moreover, the substitutions are done literally, meaning that the operand expression is directly copied into the places of substitution.

Thus, a nested application

$$\underbrace{@@\ldots@}_{n}\ \lambda u_1 \ldots \lambda u_n\ e_0\ e_1 \ldots e_n$$

effects n β-reductions in sequence, of which the i-th traverses the abstraction body $\lambda u_{i+1} \ldots \lambda u_n\ e_0'$ in search for the bound variable u_i and copies, again by a traversal, the operand expression e_i into e_0' as many times as there are free occurrences of u_i.[3]

To get the scoping of variables right, the machine must, while traversing the body expression, memorize how many more abstractors λu_i have been crossed, and compare them with the protection keys by which occurrences of u_i are preceded in order to decide whether to substitute, to take one key away or to do nothing. Moreover, the crossings of all other abstractors must be kept track of as well since copying an operand expression e_i into some place inside e_0 may require protecting free variable occurrences in e_i against possible parasitic bindings. This is routinely done irrespective of whether or not naming conflicts do indeed occur. Thus, the implementation of the β-reduction rule is as complex as it can possibly be, which in large part must be attributed to the use of named variables.

The machine works exclusively with stacks to manipulate expression strings. A machine state is given by

$$(S,\ M,\ C;\ U,\ V,\ B,\ cc)\ ,$$

and the actions of the machine are defined by a state transition function

$$\tau_{BS} : (S,\ M,\ C;\ U,\ V,\ B,\ cc) \to (S',\ M',\ C';\ U',\ V',\ B',\ cc')\ .$$

The stacks S, M, C serve the same purpose as in the SE(M)CD machine – traversing an expression from C to S (with M as an intermediate stack for constructor symbols) in search of β- and δ-redices. The stacks U, V, B are to sideline and isolate the components of β- and δ-redices, and to build up auxiliary structures while performing the respective reductions, and cc is a control stack that keeps track of nested calls for low-level control programs that are getting involved in the execution of reduction rules.

[3] Note that e_0' emerges from e_0 by substitution of the operands $e_1, \ldots, e_{i-1}$ for free occurrences of the variables $u_1, \ldots, u_{i-1}$, respectively.

A particular flavor of this machine derives from the fact that the most fundamental operations of computing – traversing, substituting, copying and deleting – are implemented in this machine by means of the preorder traversal mechanism. This mechanism may be employed not only to traverse syntactically complete (sub)expressions from any source to any sink stack (other than the control stack), but also to copy them from one source to two sink stacks (as required for substitution), and to delete them from some source stack by simply not specifying a sink stack.

Figure 6.1 shows, in the form of stack configurations, a few characteristic phases of the process of β-reducing the application[4]

$$@\ \lambda u\ @\ \lambda u\ \lambda v\ @\ /u\ v\ w\ @\ u\ v\ \ ,$$

of which the equivalent parenthesized version is

$$(\lambda u.(\lambda u.\lambda v.(/u\ v)\ w)\ (u\ v))\ \ .$$

This process substitutes the argument $@\ u\ v$ (or $(u\ v)$) for the single free occurrence of u in the body of the outermost abstraction, thereby penetrating the scope of another two abstractors λu and λv, and returns

$$@\ \lambda u\ \lambda v\ @\ @\ /u\ /v\ v\ w$$

(or, equivalently in parenthesized form, $(\lambda u.\lambda v.((/u\ /v)\ v)\ w)$).

Beginning with the complete expression set up in C and the controls cc initialized with $trav\_cs$ (for traversal from C to S), the machine reaches the configuration shown in Fig. 6.1(a) that has the outermost applicator moved to M and the abstraction exposed on C, signifying a β-redex, whereupon control calls the mode $beta\_n$ to enter into a normal order β-reduction. Next, the machine drops the abstractor λu and the applicator @ that sits on top of M, isolates in V the variable u to be substituted, and calls the control mode $r1cb$ (configuration (b)). Under this mode, the abstraction body is traversed from C to B. While doing this, the abstractors λu and λv encountered along the way have their variables pushed onto U, and each variable occurrence that flies by is compared with the variable in V (configuration (c)). The single free occurrence $/u$ of the matching variable is replaced by a so-called binding list $\bullet\ v\ \bullet\ u\ \square$ (with $\bullet$ denoting a special binary list constructor) constructed from the actual contents of stack U. This list contains exactly those bindings that have been penetrated at this point, together with a placeholder $\square$ for the argument to be substituted (configuration (d)). The mode $r1cb$ terminates with the complete abstraction body set up in left-right transposed form in B, and with an empty stack U, as the variables that have piled up in it have been removed upon leaving the scopes of the abstractors λu and λv again, which happens when they are moved from M to B (configuration (e)). The machine then switches to another instance of the control mode $trav\_cs$ that moves

[4] Again, the stack entries are separated by ':' symbols.

$$
\begin{array}{rl}
nil \mid S & \\
@^2 : nil \mid M & \\
\lambda u : @ : \lambda u : \lambda v : @ : /u : v : w : @ : u : v : nil \mid C & \\
nil \mid U & \text{(a)} \\
nil \mid B & \\
nil \mid V & \\
beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
nil \mid M & \\
@ : \lambda u : \lambda v : @ : /u : v : w : @ : u : v : nil \mid C & \\
nil \mid U & \text{(b)} \\
nil \mid B & \\
u : nil \mid V & \\
r1cb : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
\lambda v : \lambda u : @^2 : nil \mid M & \\
@ : /u : v : w : @ : u : v : nil \mid C & \\
v : u : nil \mid U & \text{(c)} \\
nil \mid B & \\
u : nil \mid V & \\
r1cb : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
\lambda v : \lambda u : @^2 : nil \mid M & \\
w : @ : u : v : nil \mid C & \\
v : u : nil \mid U & \text{(d)} \\
@ : v : \bullet\ v\ \bullet\ u\ \Box : nil \mid B & \\
u : nil \mid V & \\
r1cb : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
nil \mid M & \\
@ : u : v : nil \mid C & \\
nil \mid U & \text{(e)} \\
@ : w : \lambda u : \lambda v : @ : v : \bullet\ v\ \bullet\ u\ \Box : nil \mid B & \\
nil \mid V & \\
trav\_cs : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow &
\end{array}
$$

Fig. 6.1. Phases of β-reducing the expression @ λu @ λu λv @ /u v w @ u v

the argument in left-right transposed form from C to S (configuration (f)). Now, everything is prepared for the substitution of the argument: the control

$$
\begin{array}{rl}
@ : v : u : nil \mid S & \\
nil \mid M & \\
nil \mid C & \\
nil \mid U & \text{(f)} \\
@ : w : \lambda u : \lambda v : @ : v : \bullet\ v\ \bullet\ u\ \Box : nil \mid B & \\
nil \mid V & \\
r2bsc : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
@ : v : u : nil \mid S & \\
@^1 : \lambda v : \lambda u : @^1 : nil \mid M & \\
v : w : nil \mid C & \\
nil \mid U & \text{(g)} \\
\bullet\ v\ \bullet\ u\ \Box : nil \mid B & \\
nil \mid V & \\
r2bsc : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
\lambda v : \lambda u : @^1 : nil \mid M & \\
@ : @ : /u : /v : v : w : nil \mid C & \\
@ : u : v : nil \mid U & \text{(h)} \\
\bullet\ v\ \bullet\ u\ \Box : nil \mid B & \\
nil \mid V & \\
r2bsc : beta\_n : trav\_cs : nil \mid cc & \\
\Downarrow & \\
nil \mid S & \\
nil \mid M & \\
@ : \lambda u : \lambda v : @ : @ : /u : /v : v : w : nil \mid C & \\
nil \mid U & \text{(i)} \\
nil \mid B & \\
nil \mid V & \\
trav\_cs : nil \mid cc &
\end{array}
$$

Fig. 6.1. Phases of β-reduction (continued)

mode $r2bsc$ takes over to traverse the abstraction body back from B to C and whenever it encounters a binding list, it traverses the argument from S into its place.[5] While doing this, free occurrences of variables in the argument are checked against the variable entries in the binding list, and a protection key is added to the former whenever there is a match (configurations (g) and (h)). The mode $r2bsc$ terminates with an abstraction body that has the bound variable correctly substituted, and thus the β-reduction completed, in C (configuration (i)). The argument copy left over in S is deleted, and the

[5] Since the traversal consumes what is in S, the argument must also be copied into U, from where the contents of S can be restored for possible further substitutions.

machine returns to the *trav_cs* mode controlled by the lowermost entry in *cc* to continue the search of further redices.

In this machine, recursion can be accomplished by means of the *Y*-combinator, using the normal order applicator @ to implement the self-application or, more directly, by means of the primitive recursion operator μ (see Sect. 4.6). The δ-reduction rules are implemented in essentially the same way as in the SE(M)CD machines.

The string reduction mechanisms inflict a runtime complexity of $O(n^2)$ for problems of size $O(n)$, a typical example being the reversal of a list of some n elements. The one-β-reduction-at-a-time mode of operation, in conjunction with the complexity of the β-reduction itself, particularly the effort that has to be devoted to dealing with named variables and with the ensuing problem of resolving potential naming conflicts, contributes a considerable factor to this complexity. This factor depends in intricate ways on the number of variables that have to be dealt with, on the lengths of the character strings, including protection keys, by which the variables are represented, and on the updates that have to be performed on the protection keys.

Though the price that has to be paid for supporting variables at the machine level seems to be considerable, there is also a benefit for doing this: the machine in fact performs high-level transformations of λ-expressions that preserve all variable names introduced by the user, save for adding or deleting protection keys. The machine may be stopped after it has performed some prespecified number of β- (and δ-) reductions, and the intermediate expressions may be readily inspected, say for validation purposes, since they can be made visible in the same syntax and with the same variables as those in which the original expressions were submitted.[6] After such a stop, the machine may resume performing more reductions until eventually normal forms are reached. During a stop, the user may even navigate freely through the intermediate expression at hand to select a certain subexpression as the focus of further reductions. The complete and correct implementation of the β-reduction rule guarantees referential transparency, meaning that the value of a subexpression remains invariant against the context in which it is reduced. More specifically, variables that are free in a chosen context (or subexpression) but bound in a larger, surrounding context maintain their binding status irrespective of the β-reductions performed in the smaller context. To put it another way, irrespective of the order in which β-reductions are performed in which parts of an expression, a unique normal form, if it exists and can be reached with the chosen order, is always guaranteed.

[6] The conversion, say, of AL expressions into the constructor syntax understood by the machine and the re-conversion of intermediate expressions into AL representations are straightforward matters.

6.2 Wadsworth's Graph Reduction Techniques

Another concept for performing β-reductions has been proposed by Wadsworth in 1971. It avoids some of the complexity of directly operating on linear representations of expressions, as Berkling's machine does. The idea is to represent λ-expressions as graphs, i.e., as structures whose inner (or constructor) nodes include pointers to subgraphs, and to realize β-reductions by copying, deleting and rearranging pointers. Reductions are fully effected within the graphs themselves; again there is no environment involved.

The terms graph and graph reduction refer to the fact that, beyond the tree structures generated by the (constructor) syntax of λ-expressions, we have the binding structures of abstractions represented by pointers as well, as a consequence of which we may have subgraphs referenced by several pointer occurrences and also cyclic references.

The basic pointer substitution mechanism is illustrated in Fig. 6.2, using as an example the β-reduction

$$(\lambda u.((f\ u)\ u)\ e_a) \rightarrow_\beta ((f\ e_a)\ e_a) \ ,$$

which, when using constructor syntax, is given as

$$@\ \lambda u\ @\ @\ f\ u\ u\ e_a \rightarrow_\beta @\ @\ f\ e_a\ e_a \ .$$

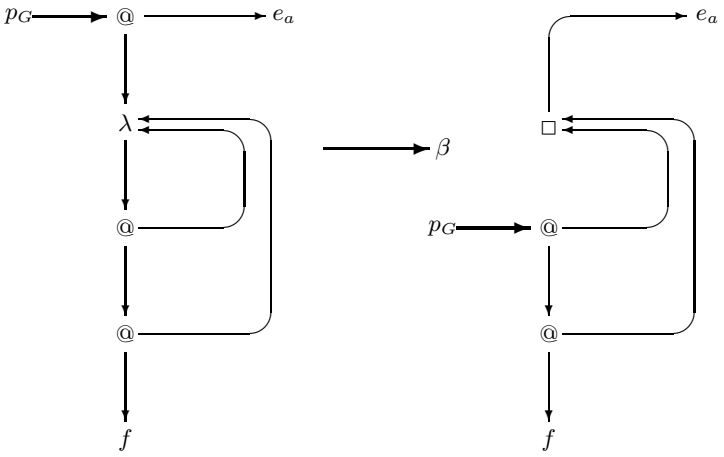


Fig. 6.2. Graph reduction Wadsworth style

The graph of the redex is shown on the left of the figure. Its topmost @-node is referenced by a graph pointer p_G; it has a left pointer (which points

downward in the graph) to its operator and a right pointer to its operand e_a. The topmost node of the abstraction graph in operator position is a λ-node representing the abstractor λu. Occurrences of the bound variable u in the body graph are 'wired up' to this λ-node by pointers that branch off to the right of the @-nodes. These pointers define the underlying binding structure in the same way as depicted in Fig. 4.3, except for their orientation in the opposite direction.

β-reduction now overwrites the λ-node with the pointer to the operand graph e_a, thus in fact establishing a substitution structure (see Fig. 4.3), and moves the graph pointer p_G down to the topmost @-node of the abstraction body. The result of this pointer rearrangement is depicted in the graph of the reductum on the right of the figure. In this graph the operand can be seen, through one level of indirection, in its two places of substitution, simply by following pointers. The single copy of the operand is thus shared between both pointer occurrences. This is the key to efficient graph reduction as it also allows sharing the evaluation of the operand expression. If, under a normal order regime, the demand for the value of an operand arises in the position of one of the pointer occurrences, this value can be made visible to all pointers directed at the shared □ node by an in-place update.

The advantage of using pointers rather than variables to connect leaf nodes to abstractor nodes is that the combined binding and substitution structures created by β-reductions remain invariant against further reductions, both in the abstraction bodies and in the operand graphs. With bound variable occurrences replaced by pointers, there can be no parasitic bindings, i.e., we have a very simple and elegant way of overcoming the problem of name clashes.[7] Variables that are free in the entire expression, such as f in our example, are treated as constant values that are not and will never get hooked up to an abstractor node.

However, performing β-reductions by such pointer rearrangements is not free of problems. Complications arise when abstractions are shared among two or more application nodes, as may be exemplified by the λ-expression

$$@\ \underbrace{\lambda x\ @\ @\ b\ x\ \lambda z\ @\ z\ a}_{operator}\ \ \underbrace{@\ \lambda x\ @\ @\ b\ x\ \lambda z\ @\ z\ a\ \lambda z\ @\ z\ a}_{operand}$$

taken from Wadsworth's thesis. Reducing the outermost redex substitutes the single occurrence of x in the body of the abstraction in operator position by the operand expression, yielding

$$@\ @\ b\ \ \underbrace{@\ \lambda x\ @\ @\ b\ x\ \lambda z\ @\ z\ a\ \lambda z\ @\ z\ a}_{operand\ substituted\ in\ abstraction\ body}\ \ \lambda z\ @\ z\ a\quad .$$

[7] There is a small trade-off involved here regarding the addressing mode used in an implementation: relative addressing requires adjustments during reduction but remains invariant against copy and move operations, whereas absolute addressing with pointers remains invariant against reduction but needs adjustments when a piece of the graph is copied or moved.

If we were to do this β-reduction by pointer manipulations as in Fig. 6.2 and were to start with a fully shared graph of the expression as depicted on the left of Fig. 6.3, then we would end up with the graph on the right. Here we have created a cyclic structure between the indirection node $\square$ that replaces the λ-node and the topmost apply node of the operand expression. This cycle is due to pointers from two different apply nodes pointing to the abstractor node of the original operator. The resulting graph referenced by p_G, therefore, is not equivalent to the expression above and is thus not correct since the cycle represents infinitely many reproductions of the operand in itself.

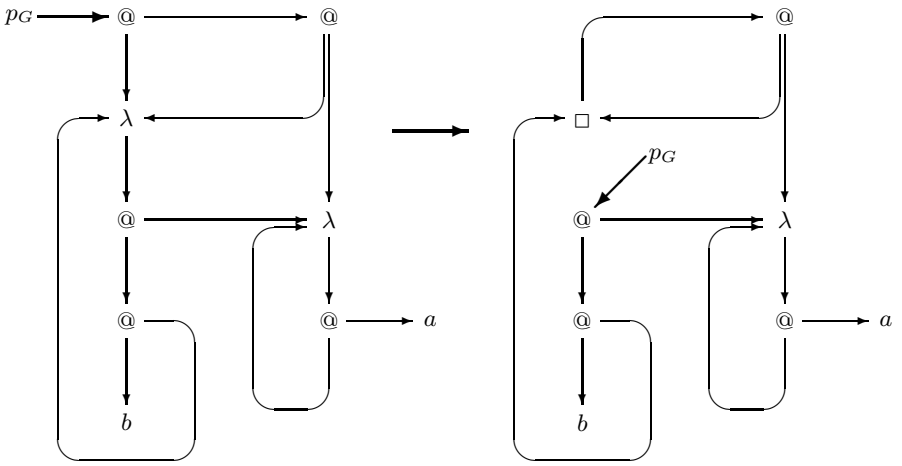


Fig. 6.3. Reducing incorrectly the fully shared graph of the expression @ λx @ @ b x λz @ z a @ λx @ @ b x λz @ z a λz @ z a

This problem can of course be fixed by making a copy of the abstraction graph so that the sharing between operator and operand of the topmost apply node is broken up, as in the top part of Fig. 6.4. The resulting graph in the lower part of the figure is now cycle-free and thus a correct graph representation of the reductum.

It should be noted that fully copying the operator to avoid sharing is a brute force measure that may introduce some redundancy. A more careful approach permits the sharing between operator and operand of maximal subgraphs in which the bound variable of the abstraction does not occur free. In our example, this would be the free variable b and the abstraction λz @ z a, in both of which we have no occurrences of x. However, it may require some effort to figure out, before β-reduction, just which parts of a graph may be

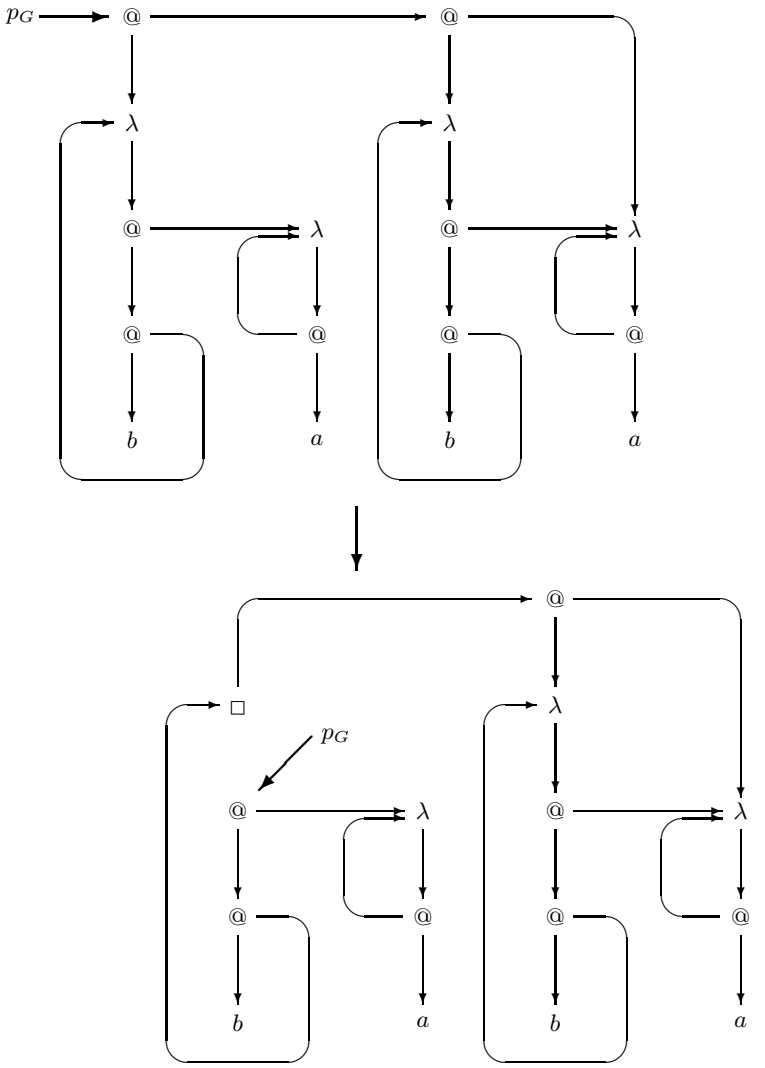


Fig. 6.4. Reducing @ λx @ @ b x λz @ z a @ λx @ @ b x λz @ z a λz @ z a with a fully copied operator graph

shared and which parts must be copied to make sure that they are correctly reduced.

Performing sequences of β-reductions on a complex graph follows a normal order regime, as defined in Sect. 4.5. The overall strategy may be considered a mix of reducing to normal form based on reducing, as far as possible, to what is called lambda form, which by Wadsworth has been referred to as RTNF/RTLF strategy. It reduces the operator graph of the top-level apply node to the

extent that an abstraction is returned as a result, if that is at all possible. This being the case, the ensuing β-redex is contracted as described, and the strategy is recursively applied to the instantiated graph of the abstraction body. Otherwise, the top-level apply node becomes part of the normalized graph, and normal order reduction continues with the operand graph. So far, the strategy is in fact weakly normalizing, which is good enough to compute full normal forms as well unless it returns abstraction graphs that cannot be applied because they are either top level or operands of apply nodes. These abstraction graphs must get their bodies recursively reduced to full normal forms. If such an abstraction is shared, the graph must be copied for in-place reduction to preserve the original graph for possible reduction in different contexts elsewhere.

It should be clearly understood that copying abstractions is the price that needs to be paid for working without an environment. The consequences of this can be painfully felt when reducing recursive functions, which are realized in the pure λ-calculus by means of the Y-combinator as

$$(Y\ f) \longmapsto_\beta (f\ (Y\ f))$$

(see Sect. 4.6). Here, replicating the abstraction graph that substitutes for f as many times as there are recursive calls must be made explicit since each instance must be reduced in a different context defined by the operands to which it is applied. Copying abstractions involves the allocation of new memory space and, along with it, constructing new instances of the binding structures – quite a space- and time-consuming undertaking that to some extent offsets the gains made by switching from string to graph reduction.

Nevertheless, the Wadsworth approach has set a standard, particularly in terms of the RTNF/RTLF strategy, for an efficient implementation of a fully normalizing λ-calculus. It can be found, in a similar form, in the machines described in the following that employ environments to avoid excessive copying.

6.3 The $\lambda\sigma$-Calculus Abstract Machine

Including an environment in a fully normalizing λ-calculus machine is not as straightforward as with weakly normalizing machines whose environments represent static mappings of either bound variables or binding indices to (values of) expressions. Neither the pairing of variables (or indices) and expressions nor the expressions themselves need ever be modified since substitutions are not allowed to penetrate abstractors and thus can be carried out naively.

As we have learned in Sects. 4.2 and 4.3, things are not so simple if we wish to perform full-fledged β-reductions. Here substitutions under abstractors require either α-converting bound variables to take them out of naming conflicts or, equivalently, updating binding indices in order to preserve correct binding structures. These corrective actions must of course carry over into the

dealings with the environment, which is nothing but a collection of delayed substitutions.

A theoretical framework that deals with this problem is provided by the λσ-calculus. This is an extension of the nameless Λ-calculus by a formal apparatus that makes environments, in the form of substitutions as defined by full-fledged β-reductions, an explicit part of it. This calculus has been invented by Abadi, Cardelli, Curien and Levy '... as a step in closing the gap between the classical λ-calculus and concrete implementations...'. The Λ-calculus of nameless dummies has been chosen for practical reasons. It eliminates the difficulties of dealing with variable names – β-reductions can be performed as relatively simple index manipulations (see Sect. 4.3) – but it also facilitates formal reasoning, say, about the preservation of the Church-Rosser property (or confluence) or the correctness of implementations.

6.3.1 The λσ-Calculus **

The syntax of the λσ-calculus (the σ stands for the part that handles substitutions) splits up into two parts, of which one covers the expressions of the Λ-calculus proper, the other the terms that describe substitutions:

$$\Lambda\text{-expressions} \quad e =_s \#i \mid @\ e_0\ e_1 \mid \Lambda e \mid e[\ s\ ]\ ,\ i \in \{0,\ 1,\ 2, \ldots\}\ ;$$

$$\text{substitutions} \quad s =_s id \mid\ \uparrow\ \mid e : s \mid s \circ t\ .$$

Both are related to each other through the expression $e[\ s\ ]$. It denotes an instance of the expression e with aggregate substitutions

$$s = \{\#0 \leftarrow e_0,\ \#1 \leftarrow e_1, \ldots\}$$

for occurrences of binding indices.[8]

The canonical forms of substitution are defined thus:

- $id = \{\#0 \leftarrow \#0,\ \#1 \leftarrow \#1, \ldots, \#i \leftarrow \#i, \ldots\}$ is the identity substitution;
- $\uparrow = \{\#0 \leftarrow \#1,\ \#1 \leftarrow \#2, \ldots, \#i \leftarrow \#(i+1), \ldots\}$ denotes index increments (or shifts by one); ;
- $\#i[\ s\ ] = s(\ \#i\ )$ is the value for $\#i$ under the substitution s;
- $e : s = \{\#0 \leftarrow e, \ldots, \#(i{+}1) \leftarrow s(\ \#i\ ), \ldots\}$ prepends e to the substitution s and increments the target indices of s by one;
- $s{\circ}t$ denotes the composition of substitutions s and t, with $e[\ s{\circ}t\ ] = e[\ s\ ][\ t\ ]$.

Substitutions in fact define environments that associate binding indices to the expressions that have to be substituted for them. The special cases id and $\uparrow$ are substitutions of the indices by themselves, which denotes an empty substitution, and by themselves incremented by one, respectively. An access to the i-th entry of the environment is denoted as $\#i[\ s\ ]$, and $e : s$ adds a

[8] As in Sect. 4.3, $\#i \leftarrow e_i$ must be read as: 'occurrences of $\#i$ are substituted by the expression e_i'.

new entry to its front end. The composition $s \circ t$ is shorthand for doing the substitution s before t. An expression $e[\ s\ ]$ is equivalent to a suspension as introduced in Sect. 5.2: it denotes the value of e in the environment s without having the substitutions actually carried out.

It should also be noted that the index increment $\uparrow$ renders binding indices other than $\#0$ superfluous since an index $\#n > \#0$ may be expressed as an n-fold application of $\uparrow$ to $\#0$, written as $\#0[\ \uparrow^n\ ]$, and that the identity substitution may also be written as $id = \uparrow^0$. Furthermore, an index $\#0$ applied to a substitution s that has e as its first element returns e, i.e., we have $\#0[\ e : s\ ] = e$. For the composition of two substitutions, of which the first is headed by the element e, we have $(\ e : s\ ) \circ t = e[\ t\ ] : (\ s \circ t\ )$.
There are two rules that distribute substitutions over composite Λ-expressions:

- the *ap*-rule $(\ @\ e_0\ e_1\ )[\ s\ ] \rightarrow_{ap} @\ e_0[\ s\ ]\ e_1[\ s\ ]$ copies a substitution s into operator and operand of an application;
- the *beta*-rule $(\Lambda e)[\ s\ ] \rightarrow_{beta} \Lambda(e[\ \#0 : (\ s \circ \uparrow\ )\ ]\ )$ has a substitution s penetrate the scope of an abstractor: it identifies all occurrences of indices in s that are free relative to Λ and increments them by one, as specified by the β-reduction rule for nameless dummies given in Sect. 4.4.

To facilitate the handling of successive β-reductions, we can define a *beta_wn*-rule as

$$@\ \Lambda e_0[\ s\ ]\ e_1 \rightarrow_{beta\_wn} e_0[\ e_1 : s\ ]\ .$$

It simply accumulates in a substitution the operands of nested β-redices as they are.[9] This rule is in fact only weakly normalizing: it does neither push substitutions over abstractors nor does it effect reductions in abstraction bodies. The more complex *beta*-rule may then only be applied whenever it becomes necessary to continue beyond weak normal forms; this may be accomplished by giving the *beta_wn*-rule precedence over the *beta*-rule.

All rules other than *ap*, *beta_wn* and *beta* that apply to substitutions are collectively called the σ-rules of the calculus. Expressions in which all substitutions have been resolved are expressions of the pure Λ-calculus. These expressions are said to be in σ-normal form.

To illustrate the application of these rules, we consider the step-by-step reduction of two simple examples.

The first example is the application[10]

$$@@\ \Lambda\Lambda\ @\ \#0\ \#1\ \#1\ \Lambda\#0\ ,$$

which after three β-reductions returns the index $\#1$. The equivalent sequence of *ap*-, *beta*(*_wn*)- and σ-rule applications is shown in Fig. 6.5.

[9] The special case $@\ \Lambda e_0\ e_1 \rightarrow_{beta\_wn} e_0[\ e_1 : id\ ]$ applies to abstractions with empty substitutions (or to abstractions that have all substitutions resolved).

[10] Note that this expression contains as the first (inner) operand an index $\#1$ that is assumed to be bound by a Λ in some surrounding context.

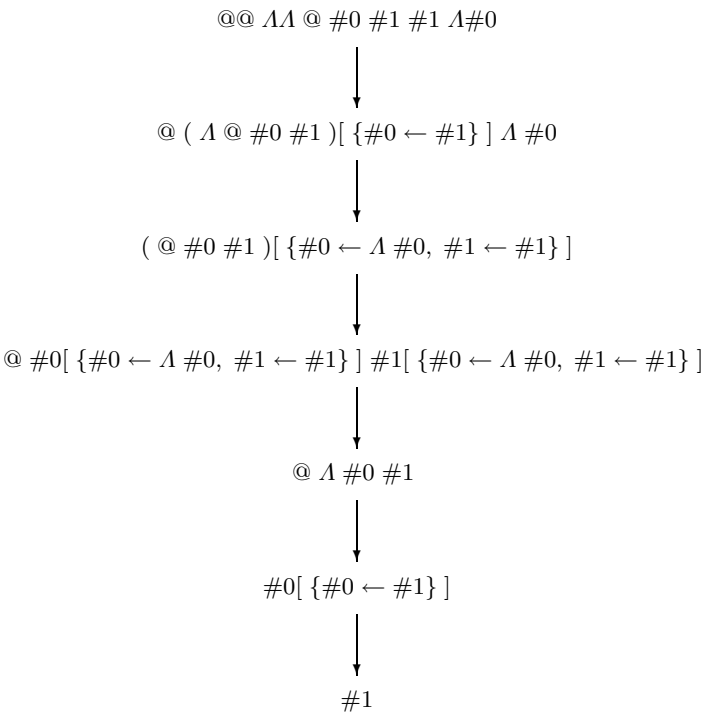


Fig. 6.5. The sequence of *beta*- and σ-reductions that normalizes the application @@ ΛΛ @ #0 #1 #1 Λ#0

The first two steps transform, by means of the *beta_wn*-rule, the two nested applications into the abstraction body @#0 #1 that has a substitution for both of its indices associated to it. The third step distributes, by application of the *ap*-rule, the substitution over the components of the application, the fourth step takes the indices that constitute the operator and the operand of the application as selectors into the substitutions to form another redex from what were originally the outer and the inner argument. At this point we have arrived at a σ-normal form since the substitutions have now been done and have disappeared from the scene. The remaining steps of the sequence apply the *beta_wn*-rule to this redex and evaluate the ensuing selector term #0[{#0 ← #1}] to #1.

Since this example features a full application, the *beta_wn*-rule suffices to compute its normal form, which in this particular case coincides with its weak normal form.

However, using the *beta*-rule, the $\lambda\sigma$-calculus can continue with reductions under abstractors should this become necessary to normalize partial applications.

The second example in Fig. 6.6 shows how this works. The expression to be reduced is the partial application @ $\Lambda\Lambda$ @ #1 #0 Λ#0, whose normal form, as can be easily verified, is Λ#0.

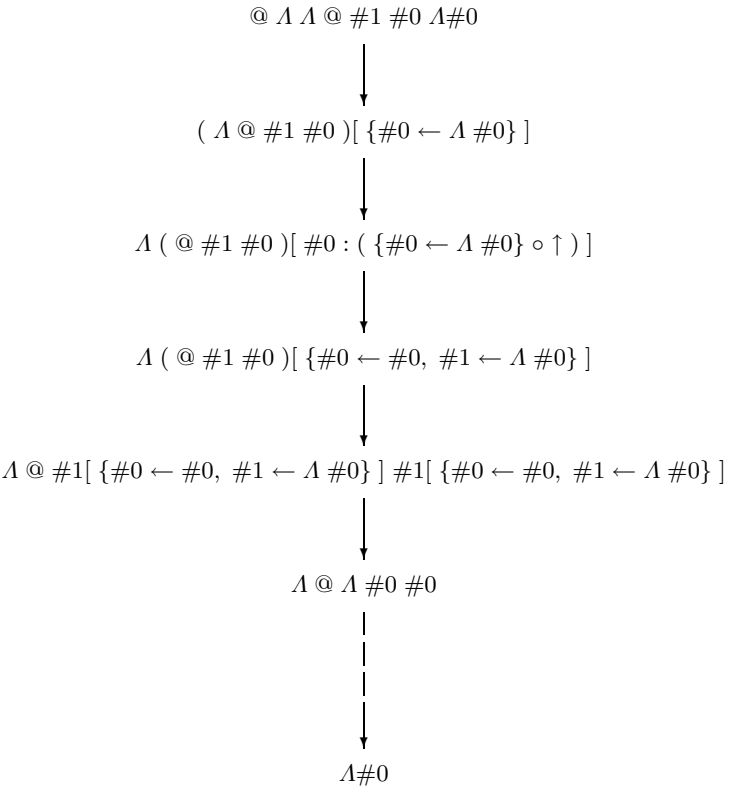


Fig. 6.6. The sequence of *beta*- and σ-reductions that normalizes the application @ $\Lambda\Lambda$ @ #1 #0 Λ#0

As the first step, the *beta_wn*-rule is called to transform the binary into a unary abstraction that has the substitution $\{\#0 \leftarrow \Lambda\#0\}$ attached to it. The second step pushes, by means of the *beta*-rule, the substitution under the abstractor and thus extends it by a new #0 entry. The following two steps prepend the new entry to, and have the shift operator $\uparrow$ update, the current substitution. Next, the new substitution is distributed, by means of the *ap*-rule, over the application that makes up the abstraction body, and the indices in the operator and operand positions select the respective entries to bring about the σ-normal form Λ @ Λ#0 #0. This becomes the starting point of another sequence of one *beta_wn*-reduction followed by two σ-reductions that terminates, as expected, with Λ #0.

6.3.2 The Abstract Machine **

We are now ready to specify an abstract machine that faithfully executes the $beta(\_wn)$- and σ-rules given in the preceding subsection. It may be considered an extension, with regard to state transition rules, of Krivine's $\mathcal{K}$-machine as it too includes just two structures T and E that hold the Λ-expression under consideration and the substitution (or environment) associated to it, respectively, and as a third structure a working stack S for operand expressions that may have to be temporarily sidelined.

Its workings are given in the usual manner by a state transition function

$$\tau_{\lambda\sigma} : (E,\ T,\ S) \to (E',\ T',\ S')$$

that maps current into next states (or configurations).

To reduce an expression e, the machine starts out with an initial state $(id,\ e,\ nil)$, i.e., with an empty substitution id and an empty stack, denoted as nil.

The state transition rules are listed in Fig. 6.7 in the order in which they need to be matched against actual configurations.

$$
\begin{array}{ll}
(1) & (\ \uparrow,\ \#i,\ S\ ) \to (\ id,\ \#(i+1),\ S\ ) \\
(2) & (\ e[\ s\ ] : t,\ \#0,\ S\ ) \to (\ s,\ e,\ S\ ) \\
(3) & (\ e : s,\ \#(i+1),\ S\ ) \to (\ s,\ \#i,\ S\ ) \\
(4) & (\ s \circ s',\ \#i,\ S\ ) \to (\ s',\ \#i[\ s\ ],\ S\ ) \\
(5) & (\ s,\ @\ e_0\ e_1,\ S\ ) \to (\ s,\ e_0,\ e_1[\ s\ ] : S\ ) \\
(6) & (\ s,\ \Lambda e_0,\ e_1 : S\ ) \to (\ e_1 : s,\ e_0,\ S\ ) \\
(7) & (\ s,\ \#i[\ id\ ],\ S\ ) \to (\ s,\ \#i,\ S\ ) \\
(8) & (\ s,\ \#i[\ \uparrow\ ],\ S\ ) \to (\ s,\ \#(i+1),\ S\ ) \\
(9) & (\ s,\ \#0[\ e : s'\ ],\ S\ ) \to (\ s,\ e,\ S\ ) \\
(10) & (\ s,\ \#(i+1)[\ e : s'\ ],\ S\ ) \to (\ s,\ \#i[\ s'\ ],\ S\ ) \\
(11) & (\ s,\ \#i[\ s' \circ s''\ ],\ S\ ) \to (\ s'' \circ s,\ \#i[\ s'\ ],\ S\ ) \\
(12) & (\ s,\ e[\ s'\ ],\ S\ ) \to (\ s' \circ s,\ e,\ S\ )
\end{array}
$$

Fig. 6.7. The state transition rules of the $\lambda\sigma$-machine

There are just two rules that manipulate the stack: rule (5) puts the operand of an application, together with the current substitution | environment, on it, and continues in T with the operator as the expression to be evaluated next, and rule (6) removes, under the control of an abstractor in T (which is also removed), the topmost expression from the stack and prepends it to the current substitution | environment, thus in fact implementing the *beta_wn*-rule.

Among the configurations in which this machine, as it has been defined, comes to a halt are two that require further action to compute full normal forms.

The first one is $(s,\ \Lambda e,\ nil)$. It has an abstraction in T that finds no argument on the stack. Together with the substitution in E, this abstraction represents a suspension (or closure) $\Lambda e[\ s\ ]$. The machine stops right there since there is no *beta*-rule that pushes the substitution over the abstractor to continue with further reductions in the abstraction body, i.e., the $\lambda\sigma$-machine is in fact only weakly normalizing.

The second one is $(id,\ \#i,\ e_1[\ s_1\ ] : \ldots : e_n[\ s_n\ ])$, for which there is no matching rule either. It represents an expression

$$\underbrace{@\ldots@}_{n}\ \#i\ e_1[\ s_1\ ]\ldots e_n[\ s_n\ ]\ ,$$

which, as we know from Sect. 4.5, is a special head normal form without leading Λs whose operand expressions (which are generally suspensions) have piled up on the stack.[11]

It takes another two rules to unlock the computation again whenever it ends up in one of these configurations. The trick in both cases is to create new contexts (or fresh $\lambda\sigma$-machines) in which the reduction may continue.

The first of these rules implements the *beta*-rule as

$$(s,\ \Lambda e,\ nil) \rightarrow \Lambda \parallel (\#0 : (\ s \circ \uparrow\ ),\ e,\ nil)\ .$$

It starts a new machine that reduces in isolation the configuration $(\#0 : (\ s \circ \uparrow\ ),\ e,\ nil)$, which represents the body of the abstraction instantiated by an updated substitution. The Λ that has been peeled off the abstraction is put in front, separated by the symbol '$\parallel$', as a reminder for the calling machine that it must be prepended to the normalized body eventually returned by the called machine in order to construct a fully normalized abstraction.

The second rule applies to the special head normal form. It starts another n fresh machines to evaluate simultaneously all n suspensions that are held in the stack:

[11] However, it should be clearly understood that this in itself is not a correct Λ-expression since there is no binder for the head index $\#i$. Such expressions come about due to the implementation of the *beta*-rule as given below, which peels leading Λs off complete head normal forms.

$$(id,\ \#i,\ e_1[\ s_1\ ] : \ldots : e_n[\ s_n\ ]\ ) \ \rightarrow\ (\#i\ \Box_1 \ldots \Box_n)\ \|\ \begin{cases} (\ s_1,\ e_1,\ nil\ ) \\ \ldots \\ (\ s_n,\ e_n,\ nil\ )\ . \end{cases}$$

Here the template $(\#i\ \Box_1 \ldots \Box_n)$ put in front tells the calling machine that an application of the general form $\underbrace{@ \ldots @}_{n}\ \#i\ e_1^{NF} \ldots e_n^{NF}$ must be constructed by substituting for the placeholders $\Box_1, \ldots,\ \Box_n$ the normal forms $e_1^{NF}, \ldots, e_n^{NF}$, respectively, returned by the called machines.

An ensemble of such machines that reduces a Λ-expression to full normal form in fact realizes a head-order reduction strategy. What may happen in general is that the *beta*-rule is called repeatedly to create a nesting of machines that perform reductions under abstractors until a head normal form is obtained by the innermost of the nested machines. This machine then turns to the operand suspensions and recursively reduces them in the same way to head normal forms as well.

Figure 6.8 shows how the $\lambda\sigma$-machine reduces the expression

$$@\ \Lambda\Lambda\ @\ \#1\ \#0\ \Lambda\#0$$

step by step to its normal form. The state transition rules that actually apply are enumerated as in Fig. 6.7. The horizontal line following the third step signifies application of the *beta*-rule that creates another machine to weakly normalize in isolation the body of the abstraction $\Lambda\ @\ \#1\ \#0$ in an updated environment that reflects its crossing over an abstractor.

The new machine terminates correctly with the value $\#0$ in the structure T, with some nonempty environment that has become irrelevant and with an empty stack, since there is no rule applicable to this configuration. The value in T is returned to the calling machine that puts in front of it the Λ that the *beta*-rule has sidelined, thus assembling the normal form $\Lambda\#0$.

6.4 Head-Order Reduction

Head-order reduction as supported by the $\lambda\sigma$-machine is a reduction strategy that combines the efficiency of the applicative order regime with full normalization. It emphasizes reduction of those subterms that are certain to contribute to normal forms, and requires simple controls. This relates to the fact that head normal forms are essential for the existence of full normal forms. The efficiency of this strategy is expected to derive largely from sharing reductions that are performed in operator (or head) positions.

In this section, we will have another look at what head-order reduction is all about, and develop an environment concept that, in contrast to the notion of explicit substitutions of the $\lambda\sigma$-calculus, directly derives from, and thus is an

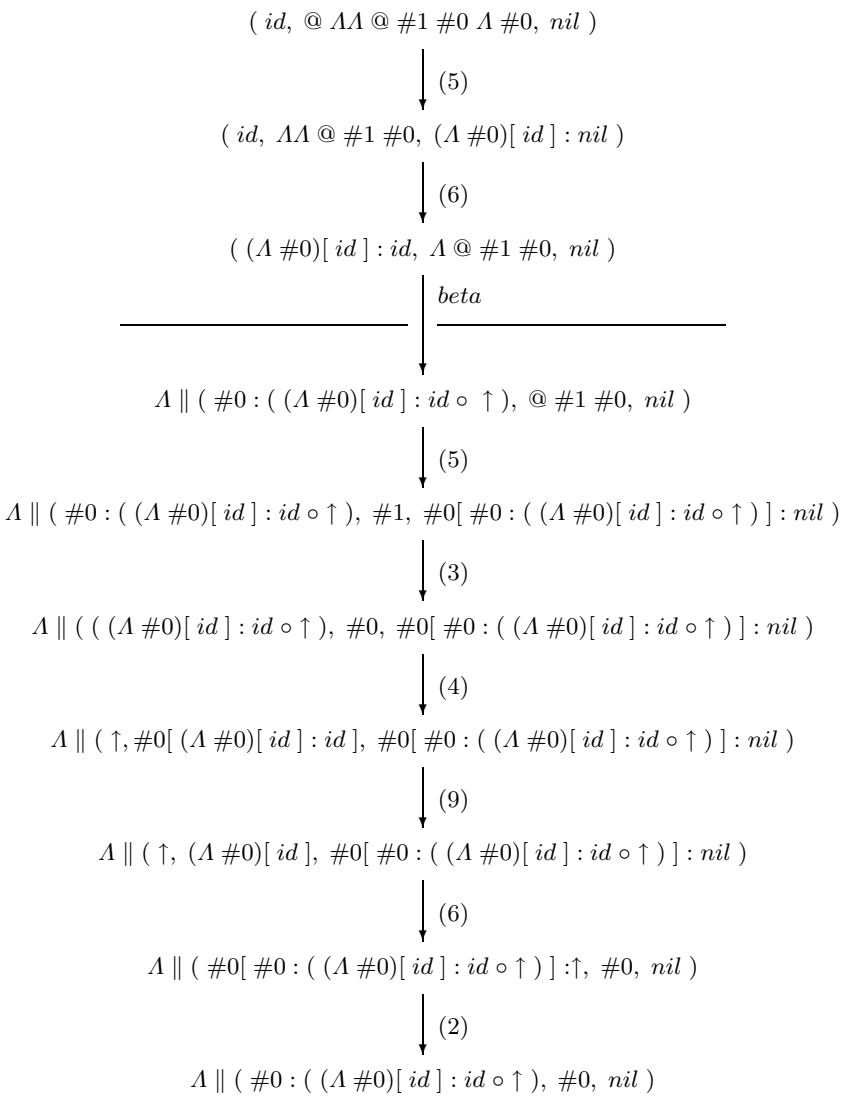


Fig. 6.8. Sequence of state transformations performed by the $\lambda\sigma$-machine when reducing the Λ-expression @ $\Lambda\Lambda$ @ #1 #0 Λ#0

integral part of, the Λ-calculus itself. We will also define, as a first step toward an implementation, an abstract machine for head-order reduction on the same level of abstraction as the $\lambda\sigma$-calculus machine. In subsequent chapters, we will move on to head-order graph reduction and introduce the mechanisms of sharing reductions in the head as a prerequisite for designing an efficient instruction-based reduction machine for a full-fledged Λ-calculus.

6.4.1 Head Forms and Head-Order β-Reductions

To convey the idea of head-order reduction as it has been developed by Berkling, and using - in slightly modified form - his easy-to-comprehend graphical representation, we simply need to look at expressions of the pure Λ-calculus in a particular way: they all feature what are called head forms that, using constructor syntax again, are generally composed of

- sequences of some $n \geq 0$ nameless abstractors Λ, followed by
- sequences of some $r \geq 0$ applicators, followed by
- a single head expression H, followed by r tail expressions T.

Head and tail expressions are recursively constructed in the same way, i.e., they all have head forms as well. The atoms are just binding indices $\#i$; they must be smaller than the number of preceding Λs since they must be bound by one of them. There is no notion of indices being free in an entire expression (see Sect. 4.3). However, indices bound to the leading sequence of Λs may be considered as being free in the sense that they are never substituted by anything but may just be passed around, and thereby updated, by β-reductions.[12] Thus, the syntax of head forms is given by

$$H \mid T =_s \#i \mid \underbrace{\Lambda \ldots \Lambda}_{n} \underbrace{@ \ \ldots \ @}_{r} \ H \ \underbrace{T \ \ldots \ T}_{r} \ .$$

If the head expression is a binding index then we have a head normal form.

A typical head form is depicted in Fig. 6.9. We recognize immediately that head and tail expressions (the latter are not shown explicitly in the figure but just indicated as thin downward pointing lines) are operators and operands, respectively, of applications. On the path from the root node down to the index $\#i$ we find alternately only sequences of Λ-abstractors and sequences of applicators @. We will refer to these sequences as *lambs* and *apps* sequences, respectively, and to the entire path as the (leftmost) spine of the head form. All tail expressions along this spine have recursively head forms, or spines, of their own.[13]

A section of the spine headed by a *lambs* sequence of length n is in fact a curried n-ary abstraction whose body stretches over the entire remaining spine, i.e., the spine of Fig. 6.9 is composed of four abstractions nested inside each other.

Normal order reduction as defined in Sect. 4.5 requires that β-redices be reduced from top to bottom along such spines. It turns head forms into head normal forms in which there are no more β-redices left along the spine, i.e., the spine is 'straightened out' and the head is a binding index.

[12] When transforming a λ-expression into an equivalent Λ-expression, only the bound variable occurrences are converted into binding indices; free variable occurrences remain unchanged and are treated as constants.

[13] Λ-nodes and apply nodes are enumerated so that one can follow up more easily what is ending up where when reducing this spine.

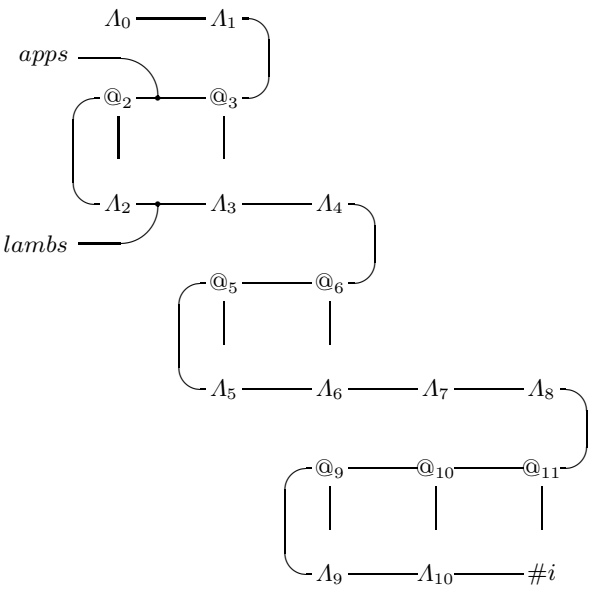


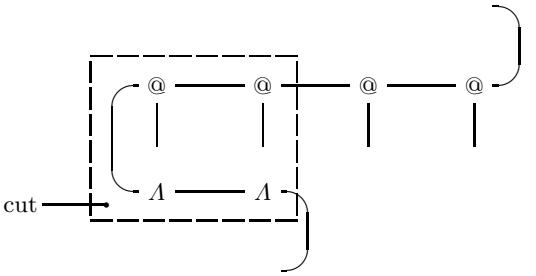
Fig. 6.9. A typical head form of a Λ-expression

In the meander-like structure of the spine in Fig. 6.9, β-redices can be found in the left-hand corners that connect *apps* and *lambs* sequences and thus pair innermost apply nodes with outermost abstractors. When reducing these redices, the *apps*–*lambs* corners shift step by step to the right, exposing the next redices, until either the abstractors or the apply nodes are exhausted. Both situations are depicted in Fig. 6.10.

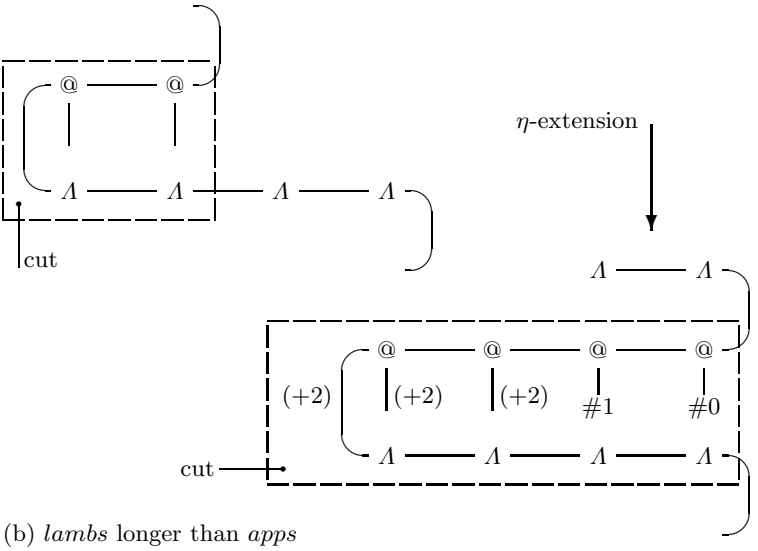
But instead of actually performing these β-reductions, head-order reduction does something different that has the effect of a delayed substitution: it simply takes largest possible numbers of β-redices, to which we will refer as cuts, off the *apps*–*lambs* corners and distributes them over the head and tail expressions of the *apps* sequence that follows next along the spine. We coin the term β-distribution_in_the_large for this operation since it distributes more than one redex in one conceptual step.

Just what the cuts are depends on the relative lengths of the *lambs* and *apps* sequences involved. The simpler case is the one shown in part (a) of Fig. 6.10. If the length of the *apps* sequence is the same as or exceeds the length of the *lambs* sequence, then we have a full application: the cut includes as many redices as there are Λ-abstractors, and the *apps* sequence that may be left over remains unaffected.

β-distribution_in_the_large is based on the semantic equivalence



(a) *apps* longer than *lambs*



(b) *lambs* longer than *apps*

Fig. 6.10. Taking cuts off left-hand corners

$$\underbrace{(\ldots(}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}.(e_a\ e_b)\ e_1)\ldots e_n) = \\ ((\underbrace{(\ldots(}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}.e_a\ e_1)\ldots e_n)\ \underbrace{(\ldots(}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ e_b\ e_1)\ldots e_n))\ ,$$

or, using constructor syntax notation,

$$\underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ @\ e_a\ e_b\ \ e_1\ldots e_n = \\ @\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ e_a\ \ e_1\ldots e_n\ \underbrace{@\ldots@}_{n}\ \underbrace{\Lambda\ldots\Lambda}_{n}\ e_b\ \ e_1\ldots e_n\ .$$

It says that the application of an n-ary abstraction to n argument expres-

sions may be distributed over the components of an abstraction body that is itself an application.

The more difficult case is the one in the lower part of Fig. 6.10, where the length of the *lambs* sequence exceeds the length of the *apps* sequence, i.e., we have a partial application that is supposed to reduce to another abstraction of arity two. When pushing the cut down the spine it needs to cross over two leftover Λ-abstractors, which means that all free occurrences of binding indices in the cut need to be incremented by two.

This is in effect equivalent to an η-extension_in_the_large of the *apps* sequence involved, before doing a β-distribution_in_the_large. Here we make use of the semantic equivalence

$$e = \underbrace{\Lambda \ldots \Lambda}_{n} \underbrace{(\ldots (}_{n} e' \ \#(n-1)) \ldots \#0) \ ,$$

or, using constructor syntax,

$$e = \underbrace{\Lambda \ldots \Lambda}_{n} \ \underbrace{@ \ldots @}_{n} \ e' \ \#(n-1) \ldots \#0 \ ,$$

where e' emerges from e when incrementing all free occurrences of binding indices by n.[14]

If we thus η-extend the *apps* sequence in part (b) of Fig. 6.10, we turn what originally was a cut that represented a partial application into a cut representing a full application We also note that the added applications have in their tails (or operand positions) the binding indices #0 and #1, and that all free occurrences of binding indices in the head and tails of the original *apps* sequence need to be stepped up by two, annotated as (+2), since two more Λ-abstractors have been squeezed in between.

β-reducing this new cut means that in the abstraction body headed by the *lambs* sequence free occurrences of the indices bound to the innermost two Λs are in fact substituted by themselves. This is of course equivalent to β-reducing the original partial application in which there were no operands to be substituted for these indices.

Figure 6.11 shows how the meander of Fig. 6.9 changes its shape when the cuts defined by *apps–lambs* corners are systematically pushed down the spine from top to bottom, using β-distributions_in_the_large and η-extensions_in_the_large, until all the cuts have accumulated in just one contiguous *apps–ambs* corner.

Beginning with the head form in the upper right, we see that the first *apps–lambs*-corner is a partial application that must be η-extended by one

[14] To convince ourselves that this is the case we simply need to look at an application $(e_0\ e_1)$ that is semantically equivalent to $(\lambda u.(e_0\ u)\ e_1)$, provided that u does not occur free in e_0, or to $(\Lambda.(e_0\ \#0)\ e_1)$.

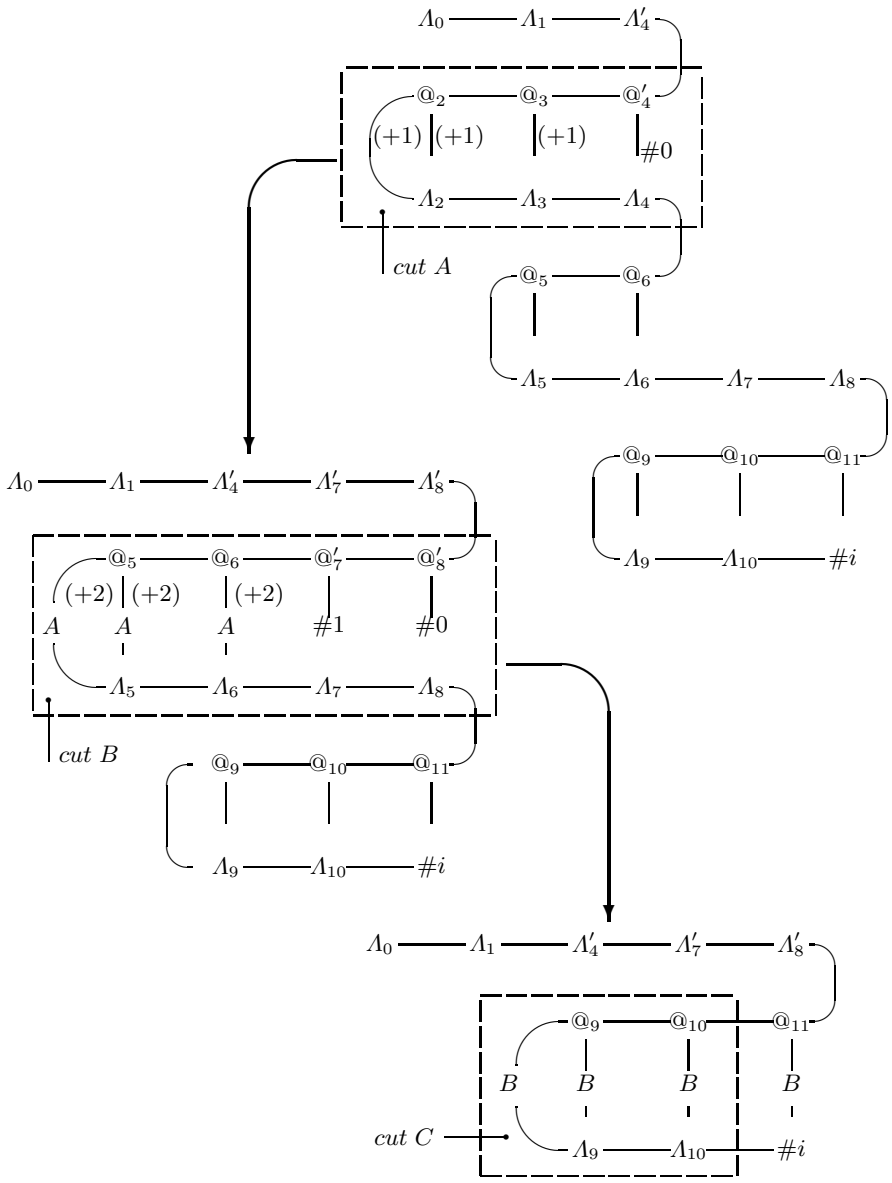


Fig. 6.11. Distributing cuts over the spine of the head form of Fig. 6.9

to obtain the cut labeled A – a full application to which the first β-distribution_in_the_large can be applied.[15] This operation pushes the cut down

[15] The apply nodes and abstractors added by η-extensions are distinguished by primes as $@'_i$ and Λ'_i, respectively.

in front of the tails of the apply nodes $@_5$ and $@_6$ and in front of the head of $@_5$ (the spine in the center of the figure). The *apps* sequence made up by these two apply nodes is part of another partial application and must therefore also be η-extended to form the cut B. Pushing this cut by another β-distribution_in_the_large down into the *apps* sequence made up from the apply nodes $@_9$, $@_{10}$ and $@_{11}$ results in a head form that has only one *apps–lambs* corner, or another cut C, left in the spine. This cut cannot be pushed any further since it ends up in front of the head index $\#i$.

However, we are not yet done. If we expand the cut B (which in turn contains the cut A) that precedes the abstraction $\Lambda_9\Lambda_{10}\,\#i$ at the lower end of the spine, we get the head form shown in Fig. 6.12 that nicely exhibits what needs to be done next.

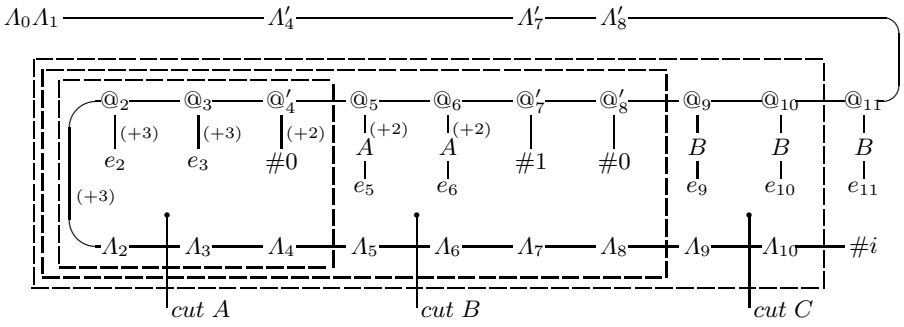


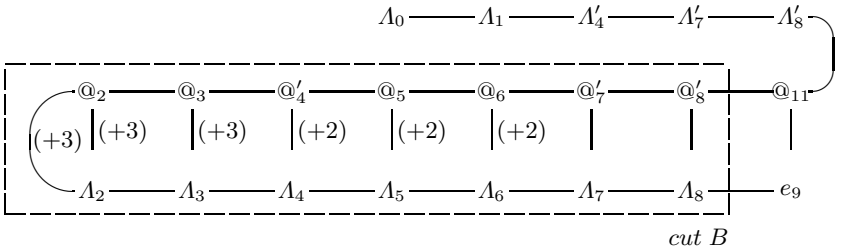
Fig. 6.12. The spine of the bottom part of Fig. 6.11, with an expanded cut B

What we have here is a spine with a single left-hand corner connecting an *apps* sequence of length ten with a *lambs* sequence of length nine, i.e., we have a nesting of nine β-redices that forms the cut C. The final operation of β-reducing these redices step by step eats up, from left to right, the entire *apps–lambs* corner, with basically two different outcomes, depending on the binding index $\#i$ at the end of the spine. This index is the entire body expression of the abstraction formed by the nine Λ-abstractors preceding it. If the index is smaller than nine, it is bound by a Λ somewhere within this *lambs* sequence, and the abstraction is in fact a selector function. When applied to the preceding *apps* sequence, it returns as the result the tail of the apply node that is in the same position relative to the corner as and opposite to the Λ to which it is bound. We call this an identity_reduction_in_the_large as it simply reproduces the selected argument.

For instance, an index $\#i = \#1$ is bound to Λ_9 and picks as the result the tail of $@_9$, and an index $\#i = \#4$ is bound to Λ_6 and selects the tail of $@_6$. These tails are appended to what is left over from the original spine, which is just the leading *lambs* sequence $\Lambda_0\Lambda_1\Lambda'_4\Lambda'_7\Lambda'_8$ followed by the apply node $@_{11}$. The new head forms are as shown in Figs. 6.13(a) and (b), respectively.

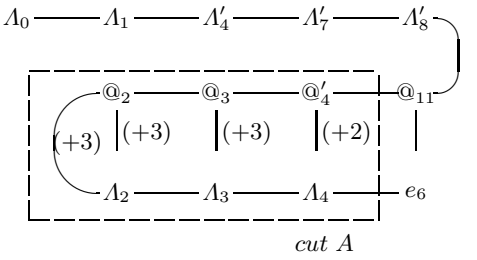
If the expressions e_9 and e_6 are non-trivial head forms themselves, we may have more β-reductions to perform along the spine thus extended.

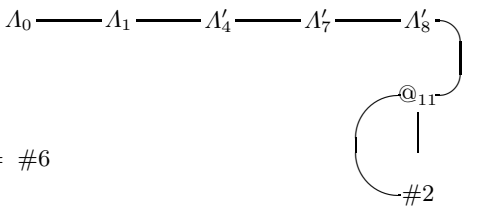
The situation is different if the index in the head position is either larger than the length of the *lambs* sequence preceding it, say $\#i = \#10$, or it is bound to one of the Λs that were subject to η-extensions, say $\#i = \#6$.



(a) selecting with index $\#i = \#1$

(b) selecting with index $\#i = \#4$





(c) selecting with index $\#i = \#6$

Fig. 6.13. Head (normal) forms resulting from β-reducing the head form of Fig. 6.12

By inspection of the original spine at the top of Fig. 6.11 we can see that the index in the former case is bound to Λ_0 and must remain so after the nine β-reductions along the *apps–lambs* corner have been performed, i.e., the correct resulting index afterwards should be $\#i = \#4$. This is indeed so since the balance between the nine intervening Λs that have been consumed by the

β-reductions and the three Λs that are added by the two η-extensions is #6, which must be subtracted from the original index value to obtain $\#i = \#4$.

The latter case is the simpler one: as the index $\#i = \#6$ is bound to Λ_4, it selects the tail of $@'_4$, i.e., the index #0 that needs to be incremented by 2 to obtain $\#i = \#2$ (see Fig. 6.13(c)).

Since in both cases there are no redices left in the head of the spine – the index there cannot be substituted by anything else – we have reached a head normal form with a remaining *apps* sequence of length one (the apply node $@_{11}$). Evaluation continues in its tail expression to recursively β-reduce it to its (head) normal form as well. Once a head normal form is reached, reducing leftover tail expressions does not change the shape of the spine anymore.

It is fairly easy to realize that the cuts that build up along the spine in fact define the environment in which the head expression is to be evaluated. This environment (or the cut) just keeps expanding as long as there are *apps–lambs* corners left to be pushed down the spine. With just one *apps–lambs* corner remaining, we have a single contiguous environment, and the head is bound to be a binding index. Depending on its value, we may either have a single access to this environment to retrieve a tail expression that must be substituted in the head, generally leading to more β-reductions along the spine, or we are done with the head and may turn to the tails, if there are any left, and recursively reduce them in head order as well.

Since the tails of head normal forms are generally unevaluated expressions preceded by cuts, or by their environments, they are equivalent to the suspensions that we know from the SE(M)CD machines of Chap. 5 and from the $\lambda\sigma$-machine of the preceding section.

The entire process composed of _in_the_large β-distributions, η-extensions and identity_reductions is organized as a sequence of what we may call β-reductions_in_the_large along leftmost spines of Λ-expressions. This process realizes a head-order reduction regime with delayed substitutions, using an environment that, conceptually, is completely contained in the framework of the Λ-calculus itself.

Not very surprisingly, there is a strong correspondence to the explicit substitution concept of the $\lambda\sigma$-calculus. The environments are of course equivalent to the $\lambda\sigma$-substitutions, β-distribution_in_the_large is equivalent to repeatedly distributing substitutions over the components of an application, η-extension_in_the_large is equivalent to repeated applications of the *beta*-rule that moves substitutions across abstractors, and identity_reduction_in_the_large is equivalent to selecting the i-th entry of a substitution.

6.4.2 An Abstract Head-Order Reduction (HOR) Machine **

An abstract machine that performs head-order reductions as outlined in the preceding subsection basically works with two structures T and E to represent the Λ-expression under consideration and the current environment, respectively. It also has a stack S for the temporary storage of tail expressions

embedded in suspensions and for other expression fragments that need to be temporarily sidelined. These are the components that the machine shares with the $\lambda\sigma$-machine and with the $\mathcal{K}$-machine. In addition, it includes a count index u that keeps track of the number of Λ-abstractors actually penetrated, to which we will also refer as the unapplied lambdas count (or ULC for short), in order to be able to adjust free occurrences of binding indices in the course of performing β-reductions, and a direction parameter dir that distinguishes between moving down a spine while reducing it to head normal form and moving up again to reduce the remaining tail suspensions from the bottom up. A complete state (or configuration) of this HOR machine is thus given by

$$(S,\ E,\ T,\ u,\ dir)\ ,$$

and the state transition rules are of the general form

$$\tau_{hor} : (S,\ E,\ T,\ u,\ dir) \rightarrow (S',\ E',\ T',\ u',\ dir')\ .$$

These rules are listed in Fig. 6.14 in the order in which they need to be tried on actual machine configurations.

In order to keep these rules close to those of the $\lambda\sigma$-machine, we abandon the notion of '_in_the_large' operations and return to individual β-reductions, η-extensions and identity reductions without changing anything conceptually. Also, instead of denoting suspensions by $e[\ s\ ]$, we use $[\ E\ e\ ]$, as in the SE(M)CD machine. The ULC index may assume values greater than or equal

$$\begin{aligned}
&(1) && (S,\ E,\ @\ e_0\ e_1,\ u,\ \downarrow) \rightarrow ([\ E\ e_1\ ] : S,\ E,\ e_0,\ u,\ \downarrow)\\
&(2) && ([\ E'\ e'\ ] : S,\ E,\ \Lambda e,\ u,\ \downarrow) \rightarrow (S,\ [\ E'\ e'\ ] : E,\ e,\ u,\ \downarrow)\\
&(3) && (S,\ E,\ \Lambda e,\ u,\ \downarrow) \rightarrow (\Lambda : S,\ (u+1) : E,\ e,\ u+1,\ \downarrow)\\
&(4) && (S,\ v : E,\ \#(i+1),\ u,\ \downarrow) \rightarrow (S,\ E,\ \#i,\ u,\ \downarrow)\\
&(5) && (S,\ [\ E'\ e\ ] : E,\ \#0,\ u,\ \downarrow) \rightarrow (S,\ E',\ e,\ u,\ \downarrow)\\
&(6) && (S,\ u' : E,\ \#0,\ u,\ \downarrow) \rightarrow (S,\ -,\ \#(u-u'),\ u,\ \uparrow)\\
&(7) && (\Lambda : S,\ -,\ e,\ u,\ \uparrow) \rightarrow (S,\ -,\ \Lambda e,\ u-1,\ \uparrow)\\
&(8) && (@ : e_0 : S,\ -,\ e_1,\ u,\ \uparrow) \rightarrow (S,\ -,\ @\ e_0\ e_1,\ u,\ \uparrow)\\
&(9) && ([\ E'\ e_1\ ] : S,\ -,\ e_0,\ u,\ \uparrow) \rightarrow (@ : e_0 : S,\ E',\ e_1,\ u,\ \downarrow)\\
&(10) && (nil,\ -,\ e,\ u,\ \uparrow) \rightarrow (-,\ -,\ e,\ -,\ done)
\end{aligned}$$

Fig. 6.14. The state transition rules of the HOR abstract machine

to zero, and the parameter *dir* may assume the values ↓ (for going down the spine), ↑ (for going up), and *done*.

Stack S serves as a temporary storage for operand expressions whose evaluation, on the way down a spine, needs to be suspended, and also to set up abstractors and apply nodes as markers (or anchor points) for the construction of normalized expressions on the way up. Environment entries are either suspensions that are moved over from S or ULCs pushed onto E whenever an abstractor is being crossed.

Rules (1) to (6) apply to the three syntactical figures of the pure Λ-calculus that may appear on top of the structure T while moving down. Applications have their operator expressions set up in T for further reductions in the current environment; the evaluation of the operand expressions is postponed by sidelining suspensions in S (rule (1)). Abstractions that find suspensions in S have them prepended to the current environment; otherwise the current ULCs, incremented by one, are pushed onto E, which is equivalent to β-distributions complemented by η-extensions. In either case, evaluation continues with the abstraction body in T (rules (2) and (3)).

A binding index greater than zero removes the topmost environment entry and decrements itself; a zero index may either find a suspension on top of E, which is set up for evaluation, or a ULC, signifying arrival at a head normal form. In the latter case the zero index is updated by subtracting from the current ULC the ULC value found on top of E, followed by a reversal of the direction from down to up (rules (4) to (6)). These rules combined in fact realize an identity reduction, or an environment lookup.

Rules (7) to (9) are to effect head-order reduction of the remaining tail suspensions and the construction of the normalized spine from the bottom up. Rule (7) prepends a Λ in S to the expression in T, rule (8) hooks up an evaluated suspension to the appropriate apply node, which can be found in S, and rule (9) sets up for evaluation a suspension found in S.

Rule (10) stops the machine after it has computed a full normal form.

Figure 6.15 shows how the HOR machine reduces step by step the Λ-expression Λ @ $\Lambda\Lambda$@ #1 #0 Λ #0 to its normal form $\Lambda\Lambda$ #0 (the numbers attached to the arrows identify the transformation rules that are applied).

Fundamental differences between the HOR machine and the $\lambda\sigma$-calculus machine primarily relate to the fact that the former reduces expressions through a succession of head normal forms to full normal forms, whereas the latter reduces only to weak (head) normal forms. This is reflected in the terminal configurations of the $\lambda\sigma$-machine, which require the creation of new machines to continue with reductions beyond weak (head) normal forms. It necessitates saving certain contexts not included in the definition of the $\lambda\sigma$-machine to assemble fully normalized expressions from the partial solutions delivered by these machines. The HOR machine handles these configurations simply by pushing onto the stack Λ and @ markers that serve as anchor points for the construction of full normal forms from the head normal forms of component expressions.

$$(nil,\ nil,\ \Lambda\ @\ \Lambda\Lambda@\ \#1\ \#0\ \Lambda\ \#0,\ 0,\ \downarrow)$$
$$\downarrow\ (3)$$
$$(\Lambda : nil,\ 1 : nil,\ \ @\ \Lambda\Lambda@\ \#1\ \#0\ \Lambda\ \#0,\ 1,\ \downarrow)$$
$$\downarrow\ (1)$$
$$([1 : nil\ \Lambda\ \#0] : \Lambda : nil,\ 1 : nil,\ \ \Lambda\Lambda@\ \#1\ \#0,\ 1,\ \downarrow)$$
$$\downarrow\ (2)$$
$$(\Lambda : nil,\ [1 : nil\ \Lambda\ \#0] : 1 : nil,\ \ \Lambda@\ \#1\ \#0,\ 1,\ \downarrow)$$
$$\downarrow\ (3)$$
$$(\Lambda : \Lambda : nil,\ 2 : [1 : nil\ \Lambda\ \#0] : 1 : nil,\ \ @\ \#1\ \#0,\ 2,\ \downarrow)$$
$$\downarrow\ (1)$$
$$([2 : [1 : nil\ \Lambda\ \#0] : 1 : nil\ \#0\ ] : \Lambda : \Lambda : nil,\ 2 : [1 : nil\ \Lambda\ \#0] : 1 : nil,\ \#1,\ 2,\ \downarrow)$$
$$\downarrow\ (4)$$
$$([2 : [1 : nil\ \Lambda\ \#0] : 1 : nil\ \#0\ ] : \Lambda : \Lambda : nil,\ [1 : nil\ \Lambda\ \#0] : 1 : nil,\ \#0,\ 2,\ \downarrow)$$
$$\downarrow\ (5)$$
$$([2 : [1 : nil\ \Lambda\ \#0] : 1 : nil\ \#0\ ] : \Lambda : \Lambda : nil,\ 1 : nil,\ \Lambda\ \#0,\ 2,\ \downarrow)$$
$$\downarrow\ (2)$$
$$(\Lambda : \Lambda : nil,\ [2 : [1 : nil\ \Lambda\ \#0] : 1 : nil\ \#0\ ] : 1 : nil,\ \#0,\ 2,\ \downarrow)$$
$$\downarrow\ (5)$$
$$(\Lambda : \Lambda : nil,\ 2 : [1 : nil\ \Lambda\ \#0] : 1 : nil,\ \#0,\ 2,\ \downarrow$$
$$\downarrow\ (6)$$
$$(\Lambda : \Lambda : nil,\ -\ ,\ \#0,\ 2,\ \uparrow)$$
$$\downarrow\ (7)\ \text{twice}$$
$$(nil,\ -\ , \Lambda\Lambda\ \#0,\ 0,\ \uparrow)$$
$$\downarrow\ (10)$$
$$(-\ ,\ -\ , \Lambda\Lambda\ \#0,\ -\ ,\ done)$$

Fig. 6.15. Sequence of state transformations of the HOR abstract machine when reducing $\Lambda\ @\ \Lambda\Lambda@\ \#1\ \#0\ \Lambda\ \#0$

More specifically, penetrating the scope of an abstractor is in the HOR machine handled by rule (3) that, on the way down, pushes the abstractor

onto S to expose the abstraction body in T for further reductions (and also increments the current ULC); the abstractor is picked up again on the way up by rule (7) to put it back in front of the normalized body. Both rules combined have the same effect as the *beta*-rule of the $\lambda\sigma$-machine that in fact creates a new machine to reduce the abstraction body in isolation, but memorizes the Λ that has been peeled off. The more frequent case where an abstraction finds a suspension on the stack is treated in the same way in both machines: the suspension is moved from S to E (rule (2) of the HOR machine and rule (6) of the $\lambda\sigma$-machine).

Suspensions left over in the tails of head normal forms are in the HOR machine taken care of sequentially on the way up. Rule (9) removes suspensions from the stack, sets them up for reduction in T and E, and pushes the respective normalized head expressions onto S, followed by a marker @. Rule (8) constructs normalized applications in T from the normalized tails and from the marker | head expression pairs held in S. Repeated application of both rules combined corresponds to having the $\lambda\sigma$-machine create as many additional machines as are necessary to reduce all remaining tails simultaneously.

Rules (1) of the HOR machine and (5) of the $\lambda\sigma$-machine distribute environments over the components of applications in the very same way; selection of an environment entry, which is handled by rules (4) to (6) in the HOR machine, is essentially captured by rules (2) to (4) and (9), (10) of the $\lambda\sigma$-machine.

6.5 Summary

This chapter has given an outline of four different conceptual approaches to designing fully normalizing λ-calculus machinery.

Two of them, a string reduction machine, of which an experimental version has actually been implemented in hardware, and a graph reduction concept, are based on the direct implementation of β-reductions. They may be considered interesting early case studies that, however, are of limited practical value since runtime efficiency in both cases leaves much to be desired. String reduction involves a lot of traversing, literal copying and deleting of expressions, which inflicts a runtime complexity of typically $O(n^2)$ for problems of sizes $O(n)$. The graph reduction approach is more efficient since binding structures are represented by pointers and β-reductions are effected by rearranging argument pointers, which in turn enables the sharing of (the evaluation of) argument graphs. However, copying complete graphs cannot be entirely avoided if abstractions need to be applied in different contexts, e.g., if they are recursively applied to changing arguments.

The other two approaches, which are the more interesting ones from an implementation point of view, accomplish full normalization by means of environment-based β-reductions. They set out with slightly different ideas of how an environment should enter the game, but the ends are essentially achieved by very similar means. Not very surprisingly, both approaches are based on the

nameless Λ-calculus since the handling of binding indices lends itself directly to simple machine operations.

The λσ-calculus introduces environments through the notion of explicit substitutions as an extension of the Λ-calculus. Substitutions are basically sets whose elements pair binding indices with the expressions by which they need to be replaced. They are subject to various operations such as identity, index increments, selection, adding another expression, composition, and distribution of substitutions over applications and abstractions.

The essence of this approach is that full normalization can be accomplished by weakly normalizing machinery that in between must call upon a special mechanism that pushes substitutions across Λ-abstractors and, in doing so, updates binding indices so as to maintain static binding structures.

Rather than extending the Λ-calculus by explicit substitutions, the concept of an environment can alternatively be directly derived from, or made an integral part of, Λ-expressions proper. The idea is to transform head forms of Λ-expressions to head normal forms by a head-order regime that employs what is called β-reductions_in_the_large.

Head forms emphasize the shapes of the leftmost spines of Λ-expressions. These spines feature alternating sequences of Λs, called *lambs*, and sequences of apply nodes @, called *apps*, followed by a binding index $\#i$ at the end of the spine. Consecutive *apps* and *lambs* sequences form *apps–ambs* corners. β-reductions_in_the_large systematically push these corners down the spine until a spine with a leading *lambs* sequence followed by a single contiguous *apps–lambs* corner followed by a head index $\#i$ emerges. This corner in fact represents an environment. The index $\#i$ either selects from this environment a suspension with which reduction continues in the head of the spine, or, if the index reaches beyond, returns an updated index, in which case the spine is in head normal form and done. Applying this head-order scheme recursively to the tails of head normal forms produces full normal forms eventually, if they exist.

The ensuing HOR machine just defines the basic runtime environment and mechanisms necessary to perform head-order reductions as outlined above, and in this respect is very similar to the λσ-machine. Besides an environment E, a structure T for expressions and a value stack S, it includes just another two parameters u and dir that respectively keep track of the number of abstractors actually penetrated and denote the direction in which a head (normal) form is traversed. Differences relate to the fact that the HOR machine reduces Λ-expressions through a succession of head normal forms to full normal forms, i.e., it is fully normalizing, whereas the λσ-machine reduces to weak (normal) forms only. Full normalization can, however, be achieved by continuing with fresh machines for subexpressions that have peeled off certain contexts in which they were called, and have appropriately updated the substitutions attached to them.

We will see in subsequent chapters how the basic concepts of the $\lambda\sigma$-machine and the HOR machine can be turned into code-executing graph reducers.

References

Berkling's string reduction machine was first published in [Ber75], a description of an experimental hardware implementation may be found in [Kge79] and [KS80]. Wadsworth's graph reducer and the RTLF/RTNF strategy are described in his PhD thesis [Wads71]. The work by Abadi, Cardelli, Curien and Levi on the $\lambda\sigma$-caluclus was published in [ACCL90], and the basic ideas of head-order reduction as outlined in section 6.4 were first described by Berkling in [Ber86]. A very nice overview of all four of these concepts is given in the PhD thesis of Troullinos [Trou93], from which is also adopted the formal definition of the head-order reduction machine. A first implementation of the head-order reduction concept may also be found in the PhD thesis of Hilton [Hil90]. For a formal definition of head-order reduction the reader may also consult Barendregt's textbook on λ-calculus [Bar84]. Backus' earlier papers on reduction systems that inspired Berkling to develop his string reducer are [Back72] and [Back73].

A theoretical contribution that relates the $\lambda\sigma$-calculus to abstract machines is a paper by Hardin, Maranget and Pagano [HMP98]. It uses a slightly modified variant of the $\lambda\sigma$-calculus as a unified framework for proving the correctness of the SECD machine [Lan64], Krivine's K-machine [Kri85], and the categorial abstract machine [CCM85/87], all of which are described in Chap. 5, and also of Cardelli's functional abstract machine [CMQ83] and its compiler [Car84]. A formal proof of the soundness and completeness of head-order reduction is given in [ZB89].

7

Interpreted Head-Order Graph Reduction

On the basis of what has been said in the preceding chapter about head-order reduction and how it can be mechanized in the HOR machine, we can now proceed to design another abstract machine that interprets graph representations of Λ-expressions, following a head-order strategy. In these graphs, which are held in a memory section called the heap, the inner nodes represent constructors and the leaf nodes represent binding indices (and also constant values, primitive operators, etc. of an applied Λ-calculus). The inner nodes, in addition to the node symbols themselves, also include pointers to subgraphs.

The benefits of using graph reduction are manifold. Traversing an expression in search of redices boils down to dereferencing pointers along spines, the environment created by β-distributions_in_the_large is made up from pointers to suspensions that combine unevaluated tail expressions with their environments. The environment may be represented as a linked list of frames that correspond to *apps–lambs* corners pushed down the spines. Different environments, of which several may coexist at some point during the evaluation of an expression, may then share common parts so that each frame exists exactly once in the entire environment structure.

The essence of this is that (sub)expressions and environments that, in a particular state of the computation are not of immediate interest and therefore need not be looked at, are hidden behind pointers, and that it is primarily pointers that may be substituted, copied and rearranged rather than the (sub)expressions or environments they represent. Since pointers have unit length, as opposed to expressions of sizes $O(n)$, the complexity of performing these elementary operations is usually cut down from $O(n)$ to $O(1)$, which typically brings down from $O(n^2)$ to $O(n)$ the complexity of graph reduction relative to that of string reduction.

The most important benefit of graph reduction derives from sharing a single copy of a suspension held in the environment among several pointers distributed over the graph. If the suspension needs to be evaluated in the place of one of these pointer occurrences, then the resulting (head) normal

form may be made visible in (or shared by) all of them by overwriting the original suspension with it.

Sharing is the key to optimizing normal order strategies with regard to numbers of reductions performed. It aims at reducing every (sub)expression at most once and, if at all, only as far as is needed to arrive at a full normal form of the entire expression graph eventually. This strategy is also referred to as lazy evaluation.

Head-order reduction looks like the best possible choice for this purpose. Under this strategy, opportunities for sharing primarily come about when suspensions are copied from the environment into head positions of spines, which is where most of the action takes place. Reducing suspensions in these positions may then be shared with the environment and thus with all pointers to the suspensions elsewhere. Tails that are left over after head-normalization are recursively treated in the same way. This strategy can be expected to compute, through a succession of head normal forms, full normal forms with least numbers of β-reductions.[1]

The head-order graph reducer whose workings will be outlined in this chapter is called the G_HOR machine as it derives more or less directly from the abstract HOR machine of Sect. 6.4.2, bringing it one step closer to a conceivable implementation.

7.1 Graph Representation and Graph Reduction

In the G_HOR machine, the graph representation in heap memory of expressions of the pure Λ-calculus is based on three types of cells for inner nodes, these being Λ-nodes, apply nodes, and suspension nodes, the latter being created in the course of reducing head forms to head normal forms. The machine also needs some suitable representation for environment frames and frame entries.

For the Λ-nodes, we take advantage of the fact that in a *lambs* sequence they follow each other in linear order, with no other graph structures branching off. A single Λ-node cell may therefore be used to represent such a sequence by a triple $(\Lambda,\ n,\ p_h)$ whose components denote from left to right the node type, the number n of Λs in the sequence, and a pointer p_h to the *apps* sequence or to a binding index that follows next along the spine.

An apply node cell represents a triple $(@,\ p_h,\ p_t)$, where @ denotes the node type, and p_h and p_t are pointers to head and tail graphs, respectively. Likewise, a suspension cell represents a triple $(sus,\ p_E,\ p_t)$, with p_t again being a pointer to a tail graph and p_E being a pointer to the environment in which the tail may have to be reduced later on.

[1] It should, however, be noted that this strategy may not necessarily be the best thing to do in terms of runtime efficiency as the overhead involved in controlling it, as we will see later on, could easily offset the gains made by saving a few reduction steps.

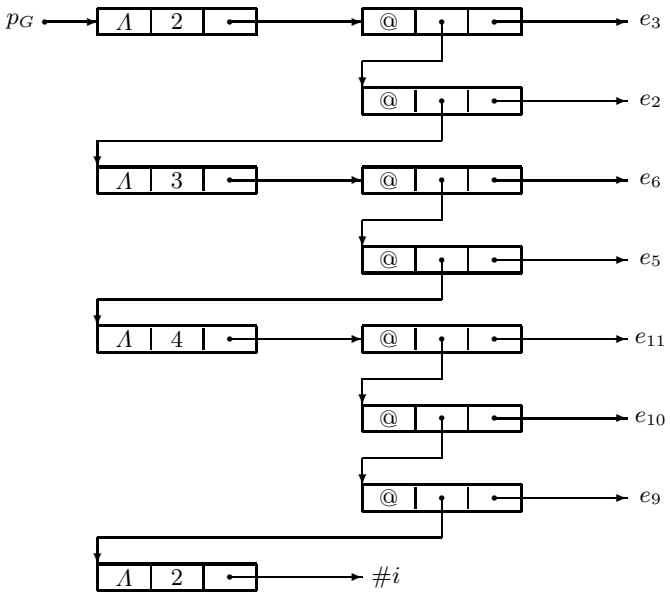


Fig. 7.1. Graph representation of the head form shown in Fig. 6.9

With these cell types at hand, the head form of Fig. 6.9 translates straightforwardly into the graph shown in Fig. 7.1. The tail graphs that are abbreviated here as $e_i \mid i \in \{ 2, 3, 5, 6, 9, 10, 11 \}$ (with the indices corresponding to the @-cells to which they are hooked up) feature similar structures as well.

The idea of reducing such graphs, roughly speaking, is to use them as nondestructible templates that are traversed from top to bottom along their spines to identify instances of β-distribution_in_the_large, to build up environments and to construct, from the bottom up, (head) normal forms as new graph structures somewhere else in the heap. The original graphs remain unchanged in this process in order to be able to reduce shared (sub)graphs in different contexts that cannot be shared.

The environment is composed of frames corresponding to *apps–lambs* corners removed from a spine that, in their order of creation, are linked up by pointers. The frames contain as many suspension entries as there are apply nodes in the particular *apps* sequences, and unapplied lambdas counts (ULCs) for missing apply nodes (or arguments), so that the total numbers of frame entries always equal the lengths of the *lambs* sequences.[2] The frame headers, in addition to the link pointers, contain as parameters the arities of the Λ-nodes and the current ULCs. When accessing the environment with specific

[2] Filling frame entries with ULCs is in fact equivalent to η-extending *apps* sequences.

head indices, these parameters are used to compute correct offsets relative to frame bases and to start evaluating the suspensions retrieved from these positions with the correct ULCs.

Reduction starts out with an empty environment, a ULC set to zero, and a graph pointer p_G pointing to the topmost (or leading) Λ-cell of the leftmost spine.The pointer to a fresh copy of this cell becomes the first entry in a trace stack that, to some extent, adopts the role of stack S of the HOR machine in that it basically keeps track of graph nodes traversed (see Sect. 6.4.2).

Very much like the instruction counter of a conventional computer, the graph pointer p_G then advances down the spine of the graph along the chain of head pointers p_h found along the way in the Λ- and apply cells.

While visiting apply cells, tail pointers are paired with the current environment pointers to form suspension cells, and the pointers to these suspensions are pushed onto the trace stack, just as specified by rule (1) of Fig. 6.14 for the HOR machine.

When encountering a Λ-cell, a new environment frame is created and filled with suspension pointers removed from the trace stack, thereby decrementing the cell's arity index, until either this index comes down to zero, in which case we have a full application, or the leading Λ-cell pops to the top of the trace stack, signifying a partial application. In this latter case, the remaining arity index is added to that of the leading Λ-cell, thus in fact lifting the unapplied Λs to top level, and the remaining entries of the frame are filled with ULCs in monotonically ascending order.[3] These operations, which correspond to removing *apps–lambs* corners from a spine, are equivalent to the HOR rules (2) and (3) of Fig. 6.14.

The suspensions left over in the trace stack after all *apps–lambs* corners have been removed and the spine has thus been straightened are the ones that constitute the tails of the resulting head normal form. They remain to be reduced when the machine moves upward again along the spine to compute a full normal form.

Figure 7.2 shows two typical phases of turning *apps–lambs* corners of the spine of Fig. 7.1 into environment frames, with the trace stack on the left and the section of spine that includes the *apps–lambs* corner actually under consideration on the right.

The first entry pushed onto the trace stack is the pointer to a fresh copy of the leading Λ-cell of arity 2. Since this Λ-cell must become the topmost node of the resulting graph, generally with an updated arity index, its pointer entry p_t is replaced by a dummy symbol $\Box$, thus in fact disconnecting the cell from the rest of the spine. This dummy symbol serves as a placeholder for the pointer to the remainder of the (head-)normalized spine that must be substituted for it later on.

The upper part of Fig. 7.2 shows the spine after the graph pointer p_G has advanced to the second Λ-cell from the top. The pointers to the suspension

[3] Note that a leading Λ-cell of arity n creates a frame filled with n ULCs only.

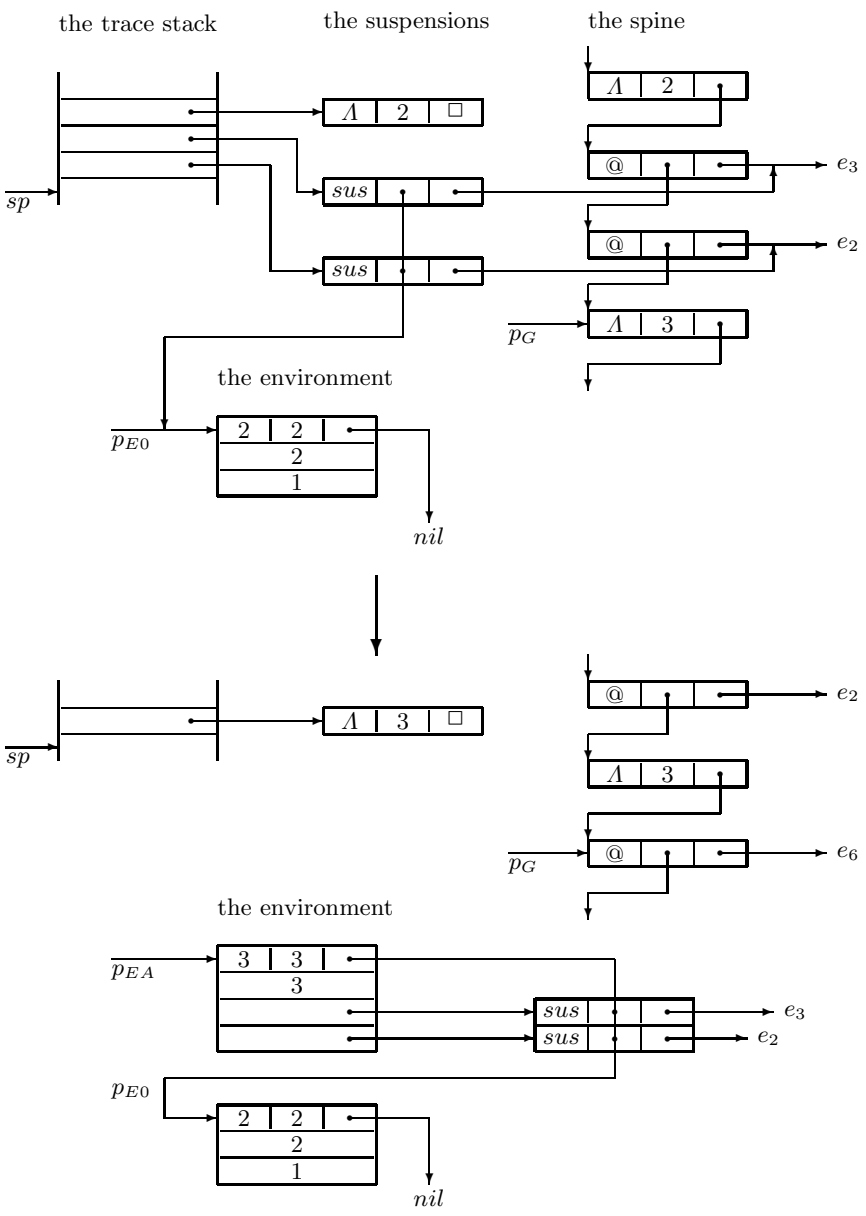


Fig. 7.2. Two phases of processing *apps–lambs* corners of the spine shown in Fig. 7.1

cells created for the two tail expressions e_3 and e_2 while the respective apply cells were visited have been pushed onto the trace stack, right on top of the

pointer to the leading Λ-cell.[4] Both suspensions share the same pointer p_{E0} to the environment created by the leading Λ-cell, which is a single frame of two ULC entries 1 and 2 prepended to an empty environment, denoted as nil.

The two suspensions now held in the trace stack belong to the first *apps–lambs* corner of the spine formed by the two apply cells just traversed and by the Λ-cell of arity 3 referenced by the current graph pointer p_G. Pushing this corner down the remainder of the spine corresponds to popping the two suspensions off the trace stack and putting them into a new environment frame, which is completed by another ULC value of 3 for the remaining unapplied Λ. This Λ also increments by one the arity index of the leading Λ-cell, which has now become the sole entry of the trace stack. The graph pointer subsequently advances to the next apply cell (with e_6 as the tail expression) down the spine. The result of these actions is shown in the lower part of Fig. 7.2.

Processing the remaining *apps–lambs* corners of the spine in the same way results in the configuration depicted in Fig. 7.3. In the upper part, this figure shows stacked up in the trace stack a single Λ-cell of arity 5 underneath a pointer to a suspension for the expression e_{11}. This expression is in operand position of an apply cell whose operator is a binding index $\#i$ referenced by p_G. The environment created at this point is shown in the lower part.

This environment consists of three frames corresponding to the cuts A, B and C in Fig. 6.12, with the pointers p_{EA}, p_{EB} and p_{EC}, respectively, pointing to them. Each frame consists of a header that includes the number n of entries, the current ULC value and a pointer to the next frame, and of the entries themselves (which are either pointers to suspensions or ULCs).

Environment accesses are performed by means of a function $lookup(\#i,\ p_E)$ that takes as its first parameter a binding index $\#i$ encountered in the head of a spine and as its second parameter the pointer p_E to the current environment. This function is expected to return either the pointer to a suspension with which the computation must continue in the head, or a ULC value that, as prescribed by HOR rule (6) of Fig. 6.14, must be subtracted from the current ULC to obtain an updated head index. In this latter case we are done with the head, i.e., we have arrived at a head normal form, and evaluation must continue with the tail suspensions of the spine that are held in the trace stack (which in Fig. 7.3 is just the suspension for the tail expression e_{11}).

Once the trace stack is empty, the evaluation has arrived at a full normal form and terminates.

In order to be able to specify *lookup* in an easy-to-read algorithmic form, we represent an environment E composed of a frame F with parameters n for the total number of its entries and u for the current ULC value and of a rest environment E' as

$$E =_s nil \mid (n,\ u,\ F) : E' \ ,$$

where

[4] The trace stack grows downward and has the stack pointer *sp* pointing to the first empty slot following the topmost entry.

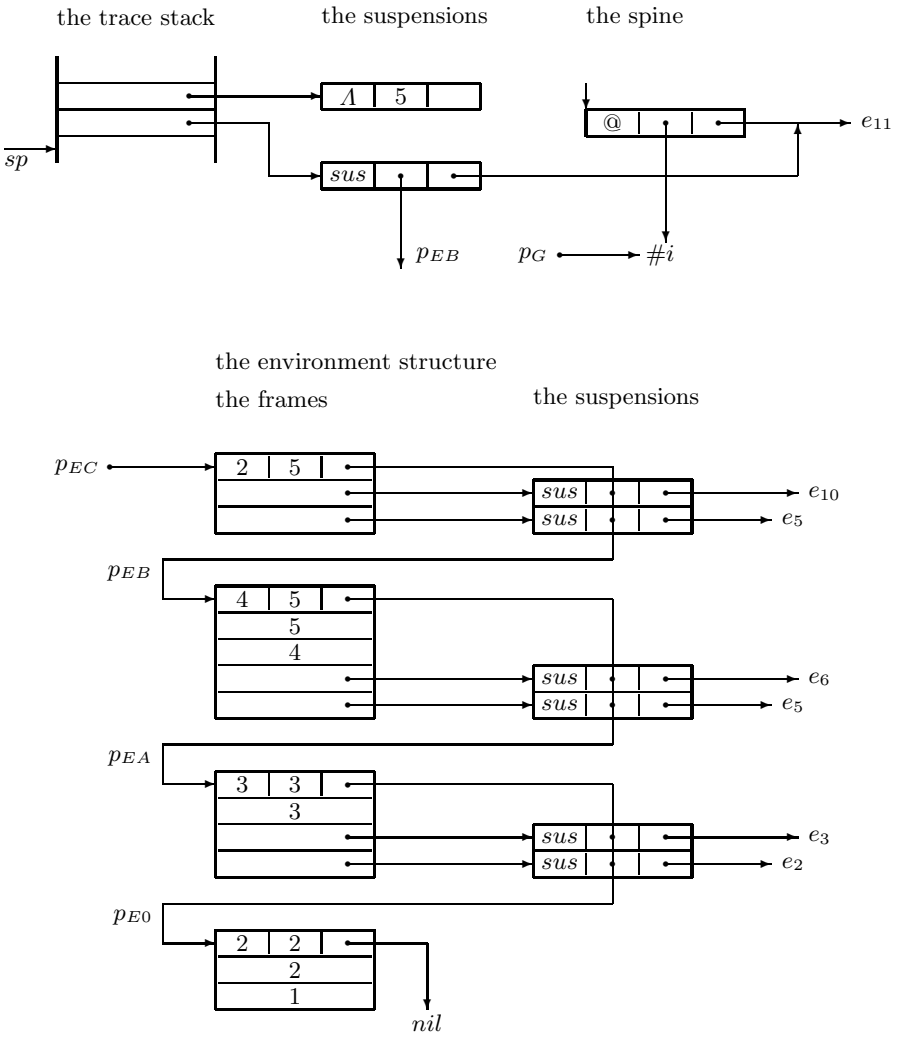


Fig. 7.3. The trace stack and the environment after the graph of Fig. 7.1 has been reduced to head normal form (the entries in the frames are arranged, from top to bottom, in the order of ascending indices)

$$F =_s F' \mid ULC : F \;\text{ and }\; F' =_s nil \mid [\, E'' \; e \,] : F' \;\;,$$

i.e., E is either empty or a sequence of frame triples $(n,\ u,\ F)$, with F being either empty or a sequence of suspensions $[\, E'' \; e \,]$ that may be preceded by a sequence of ULCs, with $ULC \in \{1,\ 2, \ldots\}$.

Using this notation, the function *lookup* may be specified as:

```
lookup( #i, E ) =
        CASE E OF
                nil  : index out of range
        (n, u, F) : E' : IF n ≥ i
                         THEN F[ i ]
                         ELSE lookup(#(i − n), E')
        END_CASE ,
```

where $F[\,i\,]$ returns the i-th entry of the frame F.

This function does the equivalent of the HOR rules (4) and (5) of Fig. 6.14. It performs, as part of a CASE statement, a pattern match on the structure of the environment E. If it is empty (denoted as nil), we have obviously an erroneous index that is out of range. Otherwise, the second pattern splits the environment E up into the components n, u and F that specify the topmost frame, and into a rest environment E'. The parameter i is compared with the frame size n and, if found smaller, meaning that it falls into the frame, the function simply returns the i-th frame entry. Otherwise, the frame size n is subtracted from i and *lookup* is recursively applied to the remaining environment E'.

7.2 Continuing with Reductions in the Head

We now have to take a closer look at what exactly needs to be done if the binding index in the head of a spine selects from the environment a suspension. Conceptually, its graph must be substituted for the head index, and reduction must then continue along its spine in the environment that comes along with the suspension.

Figure 7.4 illustrates a typical configuration that must be dealt with. In the upper part, it shows unwound on the stack a trace of two suspensions on top of a leading Λ-cell, and a graph pointer p_G that has arrived at the head index $\#i$ of the spine that is being traversed. This index is assumed to select from the current environment the suspension shown in the lower part, with p_S pointing to it. Its tail pointer p_{St} points to an as yet unreduced spine, as indicated by several Λ-nodes (which have apply nodes in between), and the associated environment is referenced by p_{SE}.

The straightforward solution to continuing in the head would be to set the graph pointer to p_{St}, the environment pointer to p_{CE}, the ULC to the value found in the header of the topmost environment frame, and then to move on.

If the graph thus substituted in the head of the current spine is, or reduces to, an abstraction, it forms a new *apps–lambs* corner with the suspensions held in the trace stack. These suspensions in fact constitute a specific instantiation (or context) of the abstraction that renders what happens further down the extended spine dependent on it, meaning that its evaluation cannot be shared. The entire process produces a (head) normal form eventually (if it exists), but

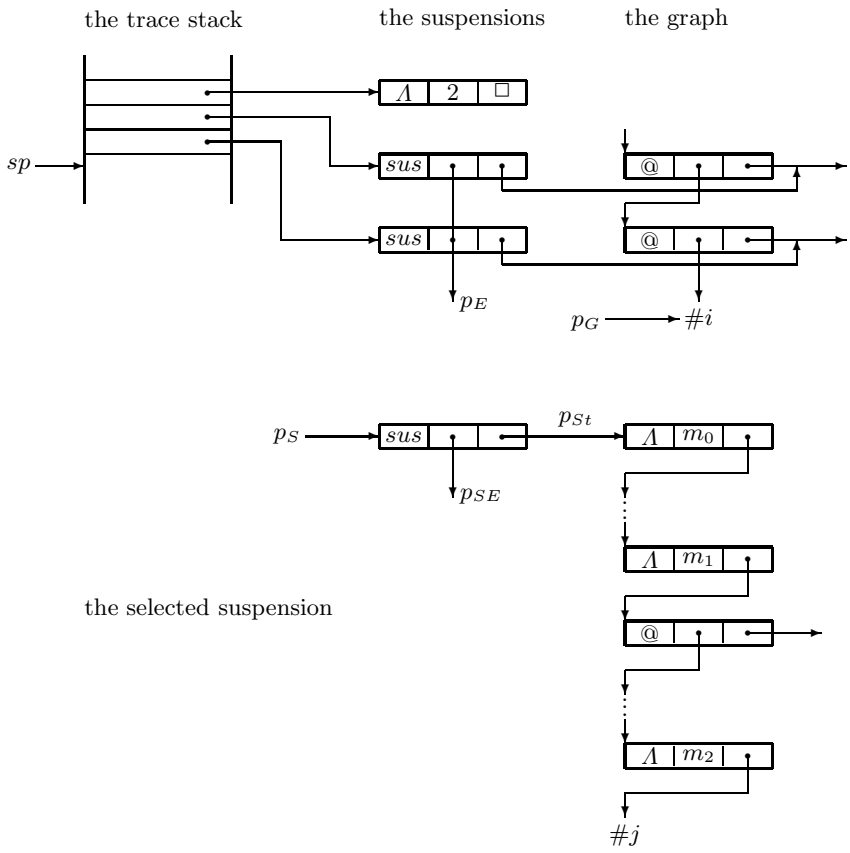


Fig. 7.4. Continuing with another suspension in the head of a spine

it may arrive there after it has reduced the same suspension more than once in different contexts, just as under an ordinary normal order regime.

Sharing makes things more complicated. It requires that the suspension selected by the head index first be evaluated in isolation, say by a fresh abstract machine, and that the graph that it returns overwrites the original suspension node so that it can be seen by all pointer occurrences directed to it. This graph may then be substituted in the head and reduction may proceed as in the nonshared case described above.

Reducing a suspension in isolation makes a lot of sense pragmatically. Even if its (head) normal form turns out to be an abstraction, it has at least all of its nonlocal parameters substituted, which may subsequently allow some simplifications of the abstraction body that could be shared with further applications. The question is just how far reducing suspensions out of context

can proceed without inflicting any problems. Unfortunately, the answer depends on what the (head) normal forms are going to be, which generally is not known *á priori*.

Reducing suspensions just to head normal forms can safely be done in isolation without regard for what we end up with. As we have learned in Sect. 4.5, their existence is essential (but by no means sufficient) to compute full normal forms eventually. Put another way: if reducing to head-normal forms does not terminate, then full normal forms do not exist either, i.e., the entire computation may be aborted right there.

However, the story would be different if the result of head normalization were an abstraction and the machine were to continue to reduce, on the way up along the spine, the tail suspensions in an attempt to compute a full normal form in isolation. Here we face the problem that some of these suspensions may be nonterminating, but that some selector that could be substituted for the head index later on, when this abstraction is applied in the intended context(s), may discard them and thus enforce termination.

This is typically the case with meaningful recursive abstractions. They include `if_then_else` clauses that in the pure Λ-calculus may have the simple head form

$$\Lambda \ldots @\ @\ \#0\ e_t\ e_f \quad ,$$

for example, in which the consequent e_t may include the recursive call and the alternative e_f may specify the terminal case. Full normalization of this expression in isolation would inevitably get trapped in unending recursions in e_t. However, if a selector abstraction $\Lambda\Lambda\ \#i \mid i \in \{0,\ 1\}$ would be substituted for the head index, it would either terminate with the value of e_f and discard e_t (if $i = 1$) or continue with e_t and drop e_f (if $i = 0$).

Of course, this termination problem would not exist if head normalization of a suspension were to return either an application or trivially a binding index, which would be identifiable as a spine headed by a Λ-cell whose arity index was zero. Then the tails could safely be reduced as well since substitution in another context would have no further effect on the shape of the spine, i.e., the normalized tails would become part of the full normal form of the entire expression. If reducing one of the tail suspensions would not terminate, then the expression would have no full normal form at all.

So, head normalization looks like the thing to do when blindly reducing suspensions out of context. A slightly smarter machine may decide to continue beyond this point if the head normal form that it has produced is something other than an abstraction, but otherwise stop right there and return the head normalized graph. But, as we will see in a moment, there are some strings attached to using head-normalized abstractions in other contexts.

The result of head-normalizing by means of a fresh machine a suspension as in Fig. 7.4 is depicted in the lower part of Fig. 7.5. This head normal form has the suspension node referenced by the pointer p_S updated by its head-normalized spine. It is composed of a leading Λ-cell of nonzero arity that

takes the place of the original suspension cell, followed by a sequence of apply cells followed by a head index. The tails of the apply cells are new suspensions constructed for the original tail expressions $e_1, \ldots, e_p$, whose environments are referenced by the pointers $p_{EE1}, \ldots, p_{EEp}$, respectively. These environments usually have more frames appended to the environment to which p_{SE} was pointing. These frames have been created in the course of running down the original spine.

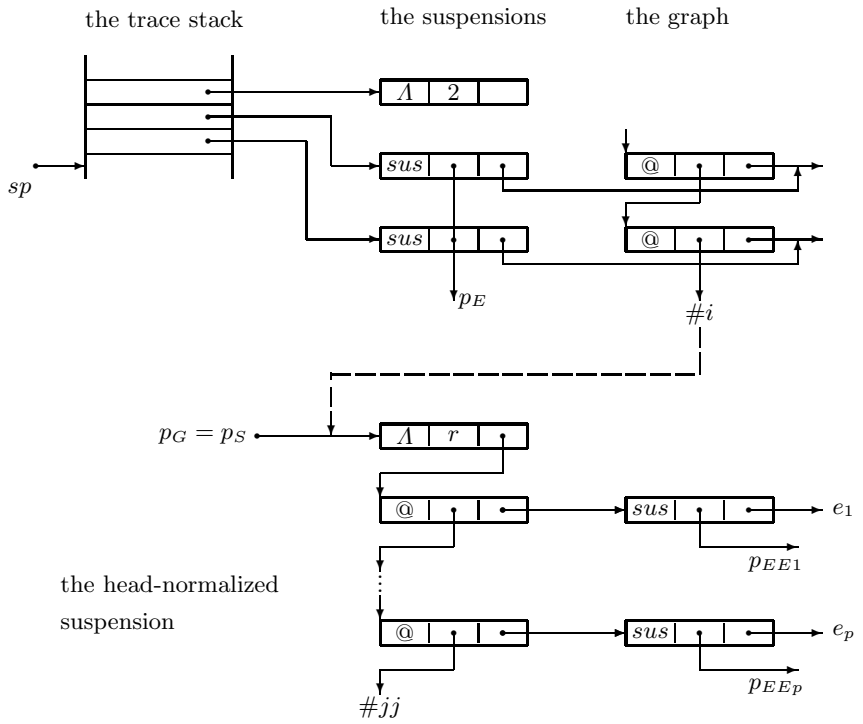


Fig. 7.5. Linking a head-normalized suspension to the head of a spine unwound on the trace stack – continued from Fig. 7.4

Wherever such a head-normalized suspension is referenced by a head index, the computation may continue, as in the nonshared case, by setting the graph pointer p_G to the updated suspension node, i.e., to p_S in the example of Fig. 7.5.

Going once more down the head-normalized spine first turns the *apps–lambs* corner formed by the Λ-cell and by the entries left in the trace stack into an argument frame, and prepends this frame to an environment consisting of just one frame that is filled in ascending order with as many ULCs as are specified by the current unapplied lambdas count (which, incidentally, equals

the arity of the Λ-cell held at the bottom of the trace stack, i.e., 2 in the example).

As an unpleasant consequence, the suspensions found in the tails of the apply cells encountered along the way must be touched again to construct new suspensions that have the original suspensions paired with this new environment. Unfortunately, these suspensions of suspensions may be passed around by further β-reductions and end up in the tails of other abstractions, which may lead to still deeper nestings of suspensions.

The real trouble starts when the machine goes into reverse gear and, on the way up, must reduce the tails in an attempt to compute a full normal form. Then these nested suspensions must be recursively unraveled from outermost to innermost in order for the machine to be able to reduce them from the inside out, each time traversing the entire graph, so that correct ULCs and hence binding indices can be preserved.

It takes little imagination to realize that the complexity thus introduced into the computation may easily offset the gains made by saving a few β-reduction steps due to sharing. But reducing in isolation suspensions to head normal form only is as far as we can go to be on the safe side of guaranteeing termination whenever a full normal form exists.

The alternatives that are available to get around the difficulties of dealing with nested suspensions are not very satisfactory either. On the one hand, we could try to reduce out-of-context suspensions to full normal form. As this entails a considerable risk of getting trapped in nonterminating recursions, it contradicts the objective of a fully normalizing machine and therefore is not acceptable. On the other hand, we could take the coward's approach of reducing suspensions to weak (head) normal forms only, leaving abstractions untouched. This strategy is in fact equivalent to full normalization if it does not end up with unapplied abstractions, but it also forecloses optimizations under abstractions that in some cases could be rather rewarding. However, in view of the difficulties with head normalization under abstractions, it may be the most reasonable thing to do.

Setting up a fresh machine to reduce in isolation a suspension either in the head or, for that matter, in one of the tails may, as in the SE(M)CD machines, be accomplished by means of a dump in which must be saved a return continuation with which the computation must be resumed upon returning. It should include at least the trace stack and the graph pointer.

The dump may have stacked up, at some instant in the course of reducing a Λ-expression, several such machines for the evaluation of nested suspensions that need to be shared.

7.3 Reducing the Tails

Once the graph pointer, on its way down the spine of a head form, has reached the binding index in its head, and this index has been replaced by an adjusted

ULC retrieved from the environment, the machine goes into reverse gear and constructs from what it finds on the trace stack the spine of the normal form. The steps that are involved in this are illustrated in Fig. 7.6.

At the top, we have a configuration that has in the trace stack two suspension pointers p_{s2} and p_{s1} on top of the pointer p_Λ to a leading Λ-cell. The current graph pointer p_{G0} points to an adjusted binding index $\#jj$ that is assumed to be correctly bound by one of the leading Λs. The normalized spine must now be constructed from the bottom up, starting with the index $\#jj$ as its leftmost leaf node. Everything else that is needed to do the job can be retrieved by dereferencing and subsequently popping pointers held in the trace stack.

Constructing a full normal form may now proceed as follows:

The suspension that is currently on top the trace stack causes the machine to construct an apply node, inserts the current graph pointer p_{G0} as its head pointer, and forces the evaluation of the suspension, i.e., in this particular case the tail expression e_{t2} in the environment $E2$, beginning with the ULC value found in the header of the topmost frame of $E2$. The resulting normal form e_{t2}^{NF} overwrites the suspension cell, and the pointer p_{s2} to it becomes the tail pointer of the apply cell just created. The pointer to this apply cell becomes the new graph pointer p_{G1}, and p_{s2} is popped off the trace stack, which brings about the second configuration from the top in Fig. 7.6.[5] These operations are equivalent to HOR rules (8) and (9) of Fig. 6.14 that handle the @ markers.

The next suspension that pops to the top of the trace stack brings about the third configuration from the top, which now features a spine of two apply nodes and an updated graph pointer p_{G2} pointing to the new node.

This piece of spine is, as the last step, hooked up to the Λ-cell now on top of the trace stack by substituting the pointer p_{G2} for the placeholder $\Box$. The pointer p_Λ is popped and taken as the resulting graph pointer, as shown at the bottom of Fig. 7.6, which completes the construction of the normalized spine. The equivalent operation of the HOR machine is specified by rule (7) of Fig. 6.14, which builds a leading *lambs* sequence from Λs held in stack S.

Constructing just a head normal form differs only insofar as the tail suspensions that are taken off the trace stack and have their pointers inserted into the newly created apply cells are left unevaluated.

As indicated in the preceding section, evaluating a suspension, whether in the head or in a tail, is done by a fresh machine that is called by saving in a dump some return continuation that includes at least the current trace stack, the graph pointer and the environment pointer.[6]

[5] Nested suspensions, other than being recursively normalized from innermost to outermost, are treated in the same way.

[6] As an aside, it should be noted that an environment becomes irrelevant when constructing a new spine from the bottom up and thus need not be saved when entering into the evaluation of a tail suspension retrieved from the stack. The

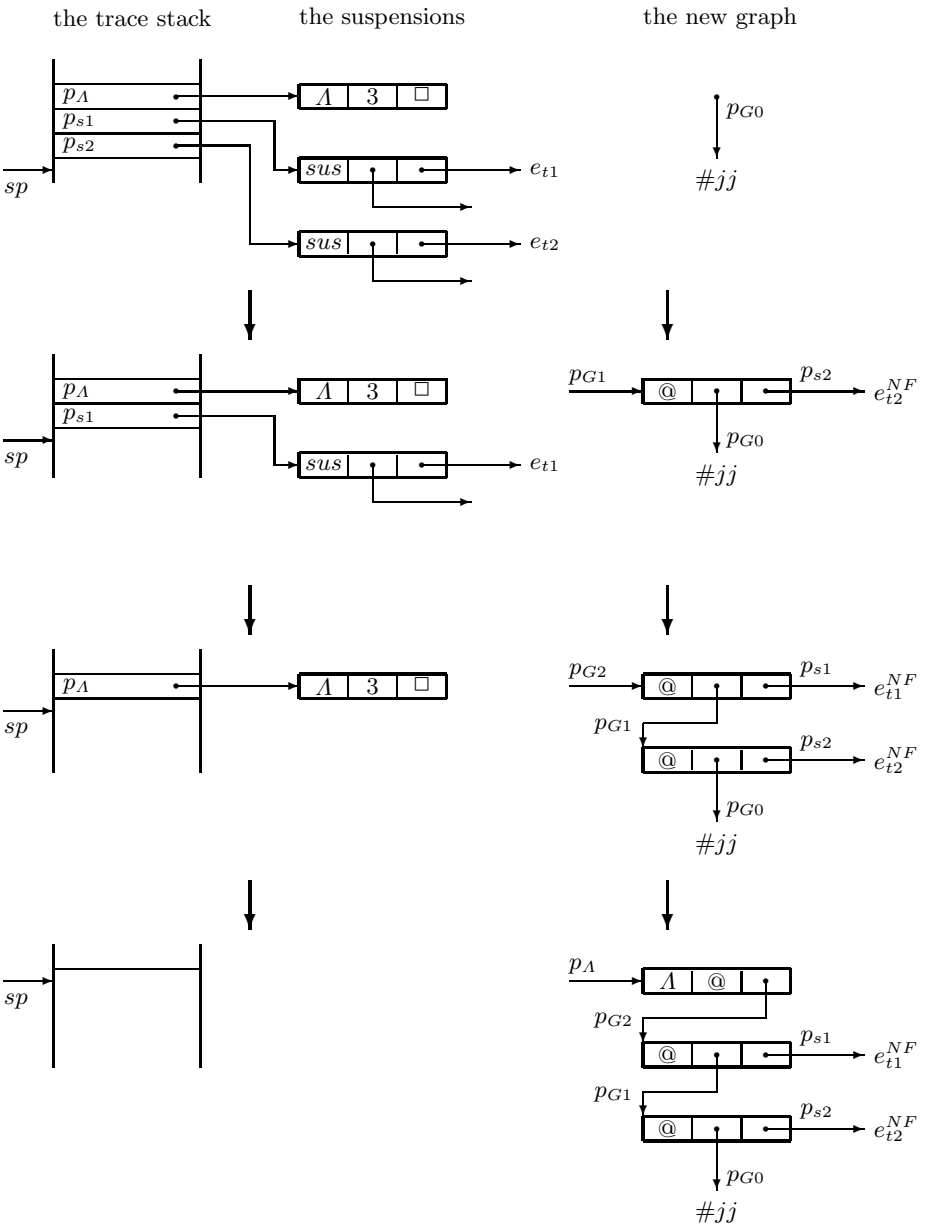


Fig. 7.6. Constructing a (head) normal form from the bottom up

environment pointer is nevertheless routinely saved since it becomes relevant when reducing a suspension in the head.

Constructing an apply node can, accordingly, be split up into two steps. The first step just creates a cell by allocating heap space, inserting the current graph pointer as the head pointer, and setting the new graph pointer to this cell. The slot for the tail pointer is, for the time being, filled with a placeholder symbol $\Box$.

Next, the evaluation of the suspension is started in a fresh machine. Upon termination, this machine returns the pointer to the normalized graph that, after the return continuation has been restored from the dump, is substituted for the placeholder in the new apply node.

7.4 An Outline of the Formal Specification of G_HOR

In the G_HOR machine, all the actions primarily take place in the heap that holds the graph and the environment. The stacks E and T of the HOR-machine are replaced by pointers into the heap, and the state transition rules predominantly specify pointer manipulations and the creation or modification of heap objects.

A state of the machine is described by the 8-tuple

$$(p_G,\ p_E,\ S,\ M,\ H,\ D,\ u,\ dir)\ ,$$

where p_G is a pointer to the graph cell that constitutes the current focus of action, p_E is the environment pointer, S is a working stack that is mainly used to build new environment frames, M is the trace stack, H denotes the heap, D denotes the dump stack, u stands for the unapplied lambdas count ULC, and dir specifies the direction in which p_G is advancing along a spine, which is either $\downarrow$ (down), $\uparrow$ (up) or *done*.

The initial configuration of the machine with which reduction of a Λ-expression e sets out, is given by

$$(p_G,\ p_E,\ nil,\ nil,\ H[\,p_G \rightsquigarrow (\Lambda,\ n,\ p_h),\ p_E \rightsquigarrow nil\,],\ nil,\ 0,\ \downarrow).$$

It has the working stack, the trace stack and the dump empty, the ULC value set to zero, the graph pointer p_G pointing to the leading Λ-cell of the graph of the expression e, p_E pointing to an as yet empty environment (denoted as nil), and the graph pointer set to move down along the spine.

The machine terminates in an orderly form with a configuration

$$(p_G,\ -,\ nil,\ nil,\ H[\,p_G \rightsquigarrow (\Lambda,\ n',\ p_{h'})\,],\ nil,\ -,\ done)\ .$$

It has the graph pointer p_G pointing to the topmost Λ-cell of the fully normalized graph, and all other structures either are empty or have become irrelevant.

The state transition rules are, as usual, specified as mappings of the general form

$$\tau_{g\_hor} : (p_G,\ p_E,\ S,\ M,\ H,\ D,\ u,\ dir) \rightarrow (p'_G,\ p'_E,\ S',\ M',\ H',\ D',\ u',\ dir')\ .$$

At this point we forgo giving the details of the fairly complex rules beyond saying that for the basic machine there are ten of them, roughly corresponding to those of the HOR machine (see Fig. 6.14), and another six rules that support sharing in the head. A full specification will be postponed until, in the next chapter, the G_HOR machine is transformed into a code-executing machine whose state transition rules can be given in a more concise form.

7.5 Garbage Collection

As we have seen throughout this chapter, the G_HOR machine, when evaluating a graph, is busy dynamically creating environment frames, suspensions and normalized pieces of graphs, for which heap space must be made available and possibly be reclaimed again if these objects are no longer needed. Managing heap space is an important aspect of implementing graph reducers. Though this is not really a primary concern of this text, there should be at least an outline of what is at stake here and how it may be done in principle.

Heap space may, beginning with an empty heap, simply be allocated in consecutive chunks in the order in which heap objects need to be created. The position at which the next object may be placed may be identified by a pointer that subsequently must be advanced by the object's size. Assuming unlimited heap space, there is nothing else to worry about, which is perfectly legitimate as long as we are talking about abstract machines. Unfortunately, heap space is finite in reality and, even if in generous supply, may get filled to capacity, at which point the computation would have to be aborted, unless something is done about heap space occupied by objects that are no longer used and may therefore be released again. Unused heap objects are also referred to as garbage, and the process of cleaning up heap space from garbage is called garbage collection.

The difficult part about garbage collection is that heap objects generally cannot be released in an order that somehow relates to the order in which they come into existence. The problem is primarily due to the entanglement between suspensions (or closures for unapplied abstractions in other settings) and environments. When removing an *apps–lambs* corner from the graph (which mimics β-distributions_in_the_large), the G_HOR-machine generally prepends another frame of suspensions to the current environment. The suspensions that, in turn, hold on to earlier environments may be passed along and copied without bounds into other contexts that are unrelated to the contexts in which they have been created. Thus, after the reduction of a modestly complex expressions has progressed to some extent, the machine typically has created a fairly complex network of pointers to environments whose entries are pointers to suspensions that in turn include pointers to (earlier) environments, and so on, with generally several pointers directed at a particular part of the environment (or, more precisely, its topmost frame) and at each suspension.

The entire environment structure develops like a cactus as depicted in Fig. 7.7, in which individual frames, represented by boxes, are linked up by pointers held in the frame headers to share common subenvironments so that exactly one copy of each frame exists. At any state during the evaluation of an expression, an entire such structure must be kept alive as the computation may in intricate ways jump back and forth among the branches of the cactus. Parts of this environment structure or individual frames can only be released if they are no longer referenced from somewhere else, e.g., by suspensions.

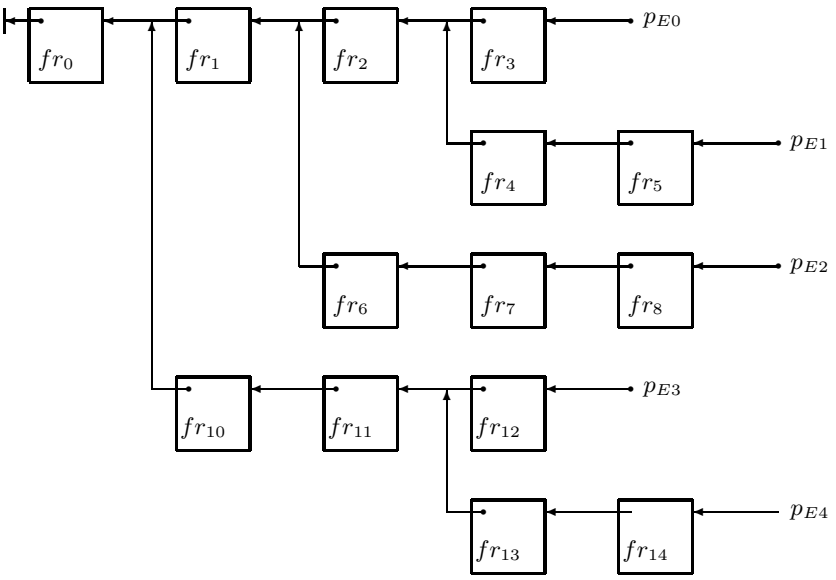


Fig. 7.7. Typical cactus structure of an environment as it develops when a Λ-expressions is reduced in head-order

Likewise, suspensions cannot be released as long as they are referenced from somewhere else, which could be environment entries or the graph.

That is to say, in order to be able to decide whether or not the space occupied by an object can be reclaimed (or garbage-collected), the system has to know or find out that no more references to it exist. Also, since heap objects are generally not released in the same or reverse order of their creation, the entire heap space tends to become fragmented with ellapsing time into occupied chunks interspersed with free chunks, also called holes, of varying sizes. The holes are usually administered by a linked list. Whenever a new heap object is about to be created, this list is searched for a hole of fitting size into which the new object can be placed. Reclaiming heap space involves adding

to this list a hole that, whenever possible, must be merged with adjacent holes in order to keep chunks of free space as large as possible.

The most rigorous approach to garbage collection is based on reference counting. It associates with every heap object a reference count value that is updated whenever a pointer to the object is replicated or consumed; once this count value comes down to zero, the space occupied by the object may be immediately released, thus keeping at a near minimum the heap space that is actually committed. However, this rigor may have to be paid for not only with the overhead of updating reference counts but also with some unpredictable time delays due to decrement operations that may have to be recursively propagated through several levels of nested substructures. If not handled properly, garbage and thus heap space may get lost forever.

With heap space in generous supply, we may take the more relaxed approach of simply letting garbage accumulate, and recover it only whenever no hole of sufficient size can be allocated. Then the garbage collector must go through all the objects in one sweep and release those with a reference count of zero.

We may even abandon reference counting altogether and instead use a mark-and-sweep scheme that ignores garbage until it becomes necessary to worry about it. The idea is to allocate heap space from a contiguous chunk until not enough is left of it to allocate another object. Only then does the garbage collector go into action to set to zero a mark bit (which replaces the reference counter) associated with each heap object. Next, the program graph and the runtime environment are searched for pointers to heap objects still in use and sets their mark bits to one. All objects whose mark bits remain zero are considered garbage and the heap space that they occupy is subsequently reclaimed.

The problem with the mark-and-sweep approach is that the computation may be suspended for considerable periods of time while garbage is collected, since the entire program graph and the entire runtime environment must be searched to find out what is actually in use and what is not. Reference counting is more selective in this regard as it touches only garbage. Garbage collection in this case is also more or less local and incremental, spreading the overhead more evenly over the entire computation and thus over time.

Irrespective of the garbage collector actually used, there is the problem of steadily increasing fragmentation of the heap space into used and unused pieces, the reason being that a newly created object hardly ever fits exactly into an allocated hole but leaves some smaller part of it over, even if – with considerably more effort – the best-fitting hole is selected. This may lead to situations where for a particular object no hole of sufficient size can be found, even though there is enough free space left in all the holes taken together. Such situations call for heap compaction that, loosely speaking, moves all the objects into a contiguous section on one side of the heap space, leaving a contiguous large hole on the other side.

The simplest but rather wasteful form of heap compaction is a two-space copying scheme. It splits the entire heap space up into two equally sized partitions, of which one is called the from-space and the other is called the to-space. The from-space is the one actually used, the to-space is the target space for compaction. Whenever fragmentation of the from-space has reached the point where space allocation fails, all heap objects still referenced are copied in one sweep into consecutive locations of the to-space. After having completed the copying and cleaned up the from-space, the two spaces are simply flipped and the computation continues out of what has now become the from-space.

Two-space copying not only leaves one half of the total heap space unused, as any other heap compaction method it also causes a considerable problem with pointers by which heap objects are referenced from within other heap objects. These pointers must all be updated to point to the new locations in the new from-space. However, this problem may, at the expense of introducing another level of indirection, be alleviated to some extent. All it takes is to have all heap objects represented by equally sized descriptors that may be placed anywhere in an array of equally sized slots, also held in the heap but outside the to- and from-spaces. Each descriptor includes a single pointer to the heap object itself, but references to the object from anywhere else are pointers to the descriptor. Moving a heap object between the from- and the to-space then entails modifying only the single pointer included in its descriptor but leaves all pointers to this descriptor, i.e., the references from other heap objects (of which there may be several), unchanged.

A more selective approach to heap compaction is called lifetime or generation scavenging. Rather than moving all the heap objects still in use each time compaction becomes necessary, the scheme distinguishes objects of different ages. It is based on the observation that heap objects are likely to be released in reverse order of their creation, i.e., the ones that enter the game at an early stage of program execution are also the ones that have the best chances of survival. Thus there are hardly any opportunities for compaction in the heap sections taken up by objects with a long lifetime. This suggests that the entire heap space be partitioned into several sections that accommodate objects of different ages (or generations), and that compaction be applied only to the youngest section(s). Also, since the majority of references point from younger to older objects, very little pointer updating needs to be done, rendering age-based scavenging the method with the least overhead of all the combined garbage collection and heap compaction schemes.

7.6 Summary

The head-order graph reducer G_HOR informally described in this chapter derives more or less directly from, and is a concretization of, the abstract head-order reducer of the preceding chapter – the HOR machine. It is a fully normalizing machine for the pure Λ-calculus that supports full-fledged β-reductions.

Graph reduction is expected to reduce the complexity of evaluating Λ-expressions relative to string reduction from typically $O(n^2)$ to $O(n)$ for problems of size $O(n)$ since it primarily substitutes, copies, moves and rearranges pointers of unit length rather than the (sub)expressions or the environments they represent. It is the key to avoiding duplicate work by reducing at most once (sub)graphs that are shared among several pointer occurrences. Owing to the underlying head-order regime, this is done only to the extent absolutely necessary to compute normal forms.

The primary strategy that is applied to this effect in the G_HOR machine is referred to as sharing in the head. The idea is to reduce in isolation to head normal form a suspension that needs to be substituted in the head position of a spine, and to subsequently overwrite the suspension with the resulting graph so that it may be seen by all pointer occurrences that refer to it. This strategy guarantees that the head normal form thus obtained is needed at least once and that, for the time being, this is as far as reduction may proceed without running the risk, in the cases where the head normal forms turn out to be abstractions, of getting trapped, due to missing contexts, in nonterminating recursions. Once these contexts have been made available in the places of substitution, the fully instantiated head normal forms may safely be reduced to full normal forms. Non-termination can then only occur if no normal forms exist at all at all.

Unfortunately, head normalization of shared suspensions in isolation comes at the expense of needing to create nested suspensions that must be evaluated recursively from the inside out, thus inflicting a considerable overhead that may more than offset the gains made by saving a few β-reductions.

There are, however, alternatives. On the one hand, reduction could safely proceed beyond head normal forms should they turn out to be something other than abstractions, which a smart machine could easily recognize, because then the tails, if there are any, are bound to contribute to the overall full normal form and therefore have to be reduced in any case. On the other hand, we could settle for a weakly normalizing strategy that does not penetrate abstractions, thus avoiding the termination problem altogether, but this would also mean forgoing optimizations that may become possible in abstraction bodies.

The graphs used by the G_HOR-machine to represent Λ-expressions are made up from inner nodes for Λ-abstractors, applicators @ and suspensions, and from leaf nodes for binding indices $\#i$. These graphs, together with the environments in which they need to be evaluated, are held in a section of memory called the heap.

The environment is realized as a linked list of frames created by applications of abstractions along leftmost spines. The frame entries are either suspensions of tail expressions or binding indices that fill in for missing arguments. Accesses to the environment are realized by means of a function *lookup* that takes as its parameters a head index and the current environment, returning either a binding index or a suspension.

The last section of the chapter gives a brief overview of ways and means of managing the heap space that accommodates the dynamically expanding and collapsing graph structure and the runtime environment, addressing primarily garbage collection by reference counting, two-space copying and generation scavenging. Though of little relevance with regard to the discussion of abstract machines, garbage collection plays an important role when it comes to implementing real graph reduction machinery.

References

The contents of this and the next chapter are based on the head-order graph reduction concept published in [Ber86], on several private communications with Berkling, volumes of hand-written notes [Ber96, Ber97], and on material contained in the PhD theses of Hilton and Troullinos [Hil90, Trou93].

Another interpreting head-order graph reducer for a fully normalizing λ-calculus is described in [RS92]. It takes the conservative approach of sharing in the head reduction to weak normal form only.

Graph reduction of the λ-calculus was first described by Wadsworth [Wads71] (see also Sect. 6.2). An excellent treatment of term and graph rewriting (reduction) may be found in the textbook by Plasmeijer and van Eekelen [PvE93].

Graph reduction is the standard implementation technique for functional languages with a lazy semantics. Prominent examples are the G-machine [Joh84] (which will be described in Chap. 9), the spineless tagless G-machine [PeSa89], the ABC machine [PvE93], the three instruction machine TIM [FW87] and Turner's SKI-machine [Tur79].

A comprehensive text on garbage collection, with numerous references to original work, is a fairly recent book by Jones [Jo99].

8

The *B*-Machine

An interpreting graph reducer such as the G_HOR machine of the preceding chapter has to find out at runtime, based on an analysis of actual machine states, what incremental state transformations need to be done in what order. This may inflict a considerable overhead since there are often several alternative state transition rules to be tested to identify the one that matches. This overhead is the more annoying as nontrivial computations specified in terms of recursive functions tend to perform repeatedly essentially the same sequences of computational steps, except for changing parameters, that an interpreting machine must each time figure out anew as if they have never happened before.

An important idea of compilation is to have the analysis of an algorithm (or of a high-level program) done only once and without actually executing it. Based on this static analysis, the compiler constructs, as a sequence of machine instructions, in the following also referred to as the code, a complete runtime schedule that can be efficiently executed with very little (or ideally with no) overhead spent on interpreting actual machine states.

These efficiency considerations raise the question of whether and to what extent a full-fledged λ-calculus could benefit from this approach as well. The categorial abstract machine briefly described in Sect. 5.5.2 shows that at least a weakly normalizing Λ-calculus can be realized by execution of compiled code. However, things appear to be a lot more difficult with full normalization. Since the λ-calculus provides considerable freedom in designing algorithms (see also Chap. 2), it is generally impossible to statically infer in all cases whether and to what extent applications can actually be reduced, e.g., with regard to matching arities of abstractions or compatible types, and to have static code deal with such fairly complicated things as dynamically changing binding indices when penetrating or removing abstractors, or as computing new (specialized) from existing functions (abstractions). This suggests that little can be accomplished by compilation alone, some amount of interpretation seems to be unavoidable.

However, if we liberate our way of thinking about instruction-based computing from the classical von Neumann approach, we may be able to come up with a concept for a code-executing machine that is at least a fairly close match to the λ-calculus.

Good starting points for this purpose are the abstract HOR machine of Sect. 6.4 and, more specifically, its graph-reducing descendant, the G_HOR machine of the preceding chapter. As already indicated in Sect. 7.1, we may consider moving the graph pointer down the spine of a head form as being similar to advancing the instruction counter of a classical computer, and accordingly we may consider the graph nodes as instructions that essentially effect the state transformations which need to be performed.

The problem with these instructions is that, when executed on the way down the spine, they are not completely done. Once the graph pointer has reached the head index of a head-normalized spine, there is a trace of things left to do when the machine reverses gear and moves upward again. The trace generated by the G_HOR machine generally consists of a Λ-node representing the accumulated leading *lambs* sequence of the spine, followed by suspended tail expressions that have not (yet) been consumed by β-reductions. These expressions still need to be normalized to compute full normal forms. Implementing this trace in a code-executing Λ-calculus machine would have to be accomplished by a sequence of instructions that, in a way, would be complementary to, and may have to be created dynamically by, the instructions encountered while running down the spine. These complementary instructions essentially are to force the evaluation, in reverse order, of the leftover suspensions by code-controlled head-order reduction of the respective tail expressions.

We will also adopt from the G_HOR machine the idea of sharing reductions in the head as the basic mechanism for avoiding, whenever possible, duplicate work. This mechanism again employs fresh machines to reduce suspensions that are called in head positions to the code equivalents of head normal forms. These codes are, by updating, subsequently made visible to all pointer occurrences that refer to the original suspensions.

The concept of such a pure Λ-calculus machine was proposed by Berkling in response to difficulties he encountered in cleanly and efficiently resolving naming conflicts among all-quantified variables in mechanized Horn-clause resolution. This machine works with two instruction streams that mutually call each other to construct cooperatively code that can be straightforwardly converted into a high-level representation of the normal form of the Λ-expression submitted for execution.

As a reference to its inventor, we have chosen to call this machine the B-machine but taken the liberty of including a few modifications and using a slightly different notation to make it fit with the concepts and machines discussed earlier and later on in this text.

8.1 The Operating Principles of the *B*-Machine

The B-machine basically centers around two code structures, denoted as F and B. They accommodate the instruction streams that must be executed in what we from now on call the forward and backward directions (instead of downward and upward along the spine), respectively. The structure F holds the code equivalent of the spine actually processed. It dynamically generates in the structure B the code that, in turn, constructs in F the normalized spine, again as code.

The code structures F and B in fact replace the graph and the trace stack M, respectively, of the G_HOR machine. Other than that, we have again a heap H that holds pieces of code, suspension nodes and environment frames, a workspace stack S, a dump D that stacks up return continuations when entering into the reduction of suspenions either in the head or in the tails, and an environment pointer p_E that, for reasons of notational convenience, is paired with the current unapplied lambdas count (or ULC) u as $(p_E \mid u)$.[1] The environment again unfolds as a cactus structure of frames, just as in the G_HOR machine.

The basic idea of the B-machine is to start computing the forward code in F that corresponds to the spine of a head form. It generates code in B that takes care of the suspensions which the G_HOR machine piles up in its trace stack. Once the code in F is exhausted, the machine executes in reverse order the code that has built up in B to compute the code equivalent of a head-normalized or fully normalized spine in F. Since this may create opportunities for further β-reductions, these steps may have to be repeated several times until an overall full normal form is reached.

The code of some nontrivial spine held in F includes pointers to the codes for its tail expressions that are held in the heap. These codes may be loaded (copied) into F for execution by dereferencing the respective pointers.

A state of the B-machine may be described by a 7-tuple

$$((p_E \mid u),\ F,\ B,\ S,\ H,\ D,\ dir)\ ,$$

and the state transitions effected by instructions may be formally specified as:

- $\tau_F : ((p_E \mid u),\ f\_instr : F,\ B,\ S,\ H,\ D,\ fw) \rightarrow$
  $(env,\ F',\ B',\ S',\ H',\ D',\ dir)$

  for instructions f_instr executed from F, and

- $\tau_B : (-,\ F,\ b\_instr : B,\ S,\ H,\ D,\ bw) \rightarrow$
  $(env,\ F',\ B',\ W',\ H',\ D',\ dir)$

[1] This notation simply makes explicit at top level the ULC parameter u held in the header of the environment frame to which p_E is pointing. Otherwise, this parameter would have to be inspected, whenever needed, by dereferencing pointers.

for instructions *b_instr* executed from B,

where fw and bw denote the respective directions, $dir \in \{fw,\ bw,\ done\}$, and env stands for either a $(p_E \mid u)$ pair or an irrelevant environment $-$.[2]

Advancing the program counter along the codes held in F and B corresponds to deleting one by one from their front ends instructions that have been executed. It considerably facilitates formalizing the idea of having new code generated from existing code, which is the essence of doing fully normalizing code-controlled β-reductions.

The initial state of the B-machine has set up in F the code for the spine of the λ-expression e to be reduced, the pointer p_E set to the base of the heap section allocated for the environment, the ULC set to zero and the direction set to fw. All other structures are empty.

The terminal state has the code for a fully normalized spine set up in F (with references to the codes for the normalized tails recursively pointing into the heap), an irrelevant environment pointer p_E, an irrelevant unapplied lambdas count ULC, some garbage left in the heap, the code structure B and the stacks S and D empty, and the direction set to *done*.

The code equivalent of a fully normalized Λ-expression as it ends up in F may be straightforwardly converted into either a graph or a high-level textual representation.

As for sharing in the head, we will focus in the following on reducing in isolation suspensions to head normal forms only, which is the most appropriate thing to do conceptually but at the same time also the approach that is most difficult to implement. As outlined in Sect. 7.2, it is as far as code execution may safely proceed without risking nontermination should the suspensions reduce to abstractions, and in full contexts it guarantees computing full normal forms, if they exist at all. The simpler but unsatisfactory solutions of reducing suspensions indiscriminately either just to weak head normal forms or to full normal forms are not discussed here since the former case would rule out code optimizations under abstractions and the latter case could potentially cause termination problems (see Sect. 7.2).

8.2 The Instruction Set

To convey the basic idea of a code-executing λ-calculus machine, it suffices to equip it with just three instructions AP p, LAM n and VAR i, which correspond to the graph nodes of the G_HOR machine, yielding very dense code. They all have a forward interpretation for execution from the structure F, LAM n and AP pp also have a backward interpretation as they can be executed from the structure B as well.

[2] The environment becomes irrelevant whenever the machine enters or is in the backward mode.

Considering first the simpler variant of the B-machine that excludes sharing in the head, the forward interpretation of the three instructions is the following:

AP p creates a suspension for code referenced by the pointer p by pairing it with the current environment pointer p_E, hides it behind a pointer as $pp \rightsquigarrow (sus,\ p_E,\ p)$, and moves itself over into B as AP pp.

LAM n, when occurring as the first instruction in a piece of code in F, is copied into B as it is. This instruction never gets involved in β-reductions but the parameter n may be incremented due to partial applications further down the spine.

Otherwise, the instruction tries to remove, from the top of B, step by step up to n consecutive AP pp instructions, pushing the pointers pp onto the stack S and each time decrementing the parameter n by one. If only some $k < n$ AP instructions can be taken out of B, then $n-k$ ULCs are pushed onto S in monotonically ascending order, and the value $n-k$ is added to the parameter m of the LAM instruction that pops to the top of B. As the last action, the LAM instruction prepends in either case the n entries pushed onto S as a new frame to the current environment, and updates the environment pointer p_E accordingly.

VAR i replaces in head positions variables turned into binding indices. It essentially uses the function *lookup* defined in Sect. 7.1 to access with index i the current environment to determine how to continue in the head of the spine. The outcome may be either a (pointer to a) suspension, in which case the code referenced in it is set up for execution in F, or an updated index obtained by subtracting from the current ULC a ULC value retrieved from the environment, in which case the machine reverses gear and executes instructions from the structure B.

The backward interpretation of the two instructions that can also be executed from B are, in the simpler nonsharing case, as follows:

AP pp forces full normalization of the suspension $pp \rightsquigarrow (sus,\ p_E,\ p_t)$ by a fresh machine, overwrites the suspension with the resulting code, and writes the instruction back into F.

LAM n, which can only be encountered as the last instruction in B, moves itself back into F to terminate construction of the code for a (head-)normalized spine.

Sharing in the head basically requires a more sophisticated interpretation of the head instruction VAR i as it must distinguish plain suspensions, suspensions of suspensions, head-normalized code and ULCs retrieved from the environment (see Sect. 7.2). Essentially the same applies to the instruction AP pp when executed from B as its parameter pp may refer to a (nested) suspension or to head-normalized code, both of which must be fully normalized.

It should be noted that, owing to the one-to-one correspondence to graph nodes, the interpretation of the chosen B-machine instructions is about as

complex, in terms of machine configurations that need to be distinguished, as the graph-interpreting G_HOR machine of the preceding chapter, i.e., not much seems to have been gained in terms of runtime efficiency. However, a great deal of analyzing these configurations to find out what exactly must be done can be eliminated by clever compilation to code that uses more specialized instructions.

For instance, the instruction LAM n could be replaced by specialized versions LLAM n, FLAM n and PLAM n k that deal with leading Λs, full applications (to n parameters) and partial applications (to k of n parameters), respectively, if opportunities for using them can safely be inferred by static program analysis. Otherwise, the instruction LAM must be used as defined above. Essentially the same applies to the instruction VAR i. It may be replaced by a branch instruction BRA pp that transfers control directly to the evaluation of a suspension referenced by pp, or by a HVAR ii instruction signifying an updated head index in a head-normalized spine, if this can be figured out statically.

We could also think of these instructions as being macros that expand to standardized sequences of simpler conventional instructions which move things from one place to another, do primitive arithmetic | logical or relational operations, dereference pointers, and do (un)conditional branches. The ensuing codes may give rise to further optimizations known from conventional compiler technology.

8.2.1 Instruction Interpretation Without Sharing **

Following the informal description of what the B-machine instructions are supposed to do, we may now take a closer look at their formal specification, beginning in Fig. 8.1 with the state transition rules for the nonshared versions.

For operations involving the heap, we use in these rules the following notations:

- $p \rightsquigarrow object$ denotes a heap object referenced by a pointer p from somewhere else. The object itself may be a Λ-cell, an apply cell or a suspension cell, given as a triple $(\Lambda,\ n,\ p_h)$, $(@,\ p_h,\ p_t)$ and $(sus,\ (p_E \mid u),\ p_t)$, respectively, a piece of code for an expression e, denoted as $code[e]$, or an environment frame.
- $< n,\ u,\ p_E \mid s1,\ \ldots\ ,\ sn >$ represents as a list structure an environment frame composed of a header $< n,\ u,\ p_E \mid$ (with p_E pointing to the next environment frame and u denoting the ULC associated to the environment) and n frame entries $si \mid i \in \{1, \ldots, n\}$ that may be either pointers to suspensions or ULCs.
- $H[\ p \rightsquigarrow object\ ]$ says that the heap contains, generally among many others, an object $p \rightsquigarrow object$ that is actually of interest.
- $H[\ pp_1 | pp_2 \rightsquigarrow object\ ]$ says that both pp_1 and pp_2 point to the same $object$.
- $H[]$ occurring on the right-hand side of a rule means that the heap remains the same as on the left-hand side.

(1) $((p_E \mid u),\ \text{AP}\ p : F,\ B,\ S,\ H,\ D,\ fw\ ) \rightarrow$
$((p_E \mid u),\ F,\ \text{AP}\ pp : B,\ S,\ pp \rightsquigarrow (sus,\ (p_E \mid u),\ p) : H,\ D,\ fw\ )$

(2a) $((p_E \mid u),\ \text{LAM}\ n : F,\ \text{AP}\ pp : B,\ S,\ H,\ D,\ fw\ ) \mid (\ n > 0\ ) \rightarrow$
$((p_E \mid u),\ \text{LAM}\ n-1 : F,\ B,\ pp : S,\ H,\ D,\ fw\ )$

(2b) $((p_E \mid u),\ \text{LAM}\ 0 : F,\ B,\ s1 : \ldots : sm : nil,\ H,\ D,\ fw\ ) \rightarrow$
$((pp_E \mid u),\ F,\ B,\ nil,\ pp_E \rightsquigarrow < m,\ u,\ p_E \mid s1 \ldots sm >: H,\ D,\ fw\ )$

(3a) $((p_E \mid u),\ \text{LAM}\ n : F,\ nil,\ S,\ H,\ D,\ fw\ ) \rightarrow$
$((p_E \mid u),\ \text{LAM}\ n : F,\ \text{LAM}\ 0 : nil,\ S,\ H,\ D,\ fw\ )$

(3b) $((p_E \mid u),\ \text{LAM}\ n : F,\ \text{LAM}\ m : nil,\ S,\ H,\ D,\ fw\ ) \rightarrow$
$((p_E \mid u+1),\ \text{LAM}\ n-1 : F,\ \text{LAM}\ m+1 : nil,\ u+1 : S,\ H,\ D,\ fw\ )$

(5) $((p_E \mid u),\ \text{VAR}\ i : nil,\ B,\ S,$
$H[\ p_{sus} \rightsquigarrow (sus,\ (p'_E \mid u'),\ p'_t),\ p'_t \rightsquigarrow code[e']\ ],\ D,\ fw\ ) \mid$
$(\ lookup(i,\ p_E) = p_{sus}\ ) \rightarrow$
$((p'_E \mid u),\ code[e'] : nil,\ B,\ S,\ H[],\ D,\ fw\ )$

(6) $((p_E \mid u),\ \text{VAR}\ i : nil,\ B,\ S,\ H,\ D,\ fw\ ) \mid (\ lookup(i,\ p_E) = u'\ ) \rightarrow$
$(-,\ \text{VAR}\ u - u' : nil,\ B,\ S,\ H,\ D,\ bw\ )$

(7) $(-,\ ncode[e'] : nil,\ \text{LAM}\ n : nil,\ S,\ H[\ p_{sus} \rightsquigarrow (sus, \ldots)\ ],$
$(\ t,\ p_{sus},\ F,\ B,\ D\ ),\ bw\ ) \rightarrow$
$(-,\ \text{AP}\ p_{sus} : F,\ B,\ S,\ H[\ p_{sus} \rightsquigarrow \text{LAM}\ n : ncode[e']\ ],\ D,\ bw\ )$

(9) $(-,\ F,\ \text{AP}\ p_{sus} : B,\ S,$
$H[\ p_{sus} \rightsquigarrow (sus,\ (p'_E \mid u'),\ p'_t),\ p'_t \rightsquigarrow code[e']\ ],\ D,\ bw\ ) \rightarrow$
$((p'_E \mid u'),\ code[e'] : nil,\ nil,\ S,\ H[],\ (\ t,\ p_{sus},\ F,\ B,\ D\ ),\ fw\ )$

(10) $(-,\ ncode[e'] : nil,\ \text{LAM}\ n : nil,\ -,\ H,\ nil,\ bw\ ) \rightarrow$
$(-,\ \text{LAM}\ n : ncode[e'] : nil,\ -,\ H,\ nil,\ done\ )$

Fig. 8.1. The state transition rules for the basic B-machine instructions (no support for sharing in the head)

- $p \rightsquigarrow object : H[\ldots]$ occurring on the right-hand side of a rule means that a new object $p \rightsquigarrow object$ is added to the heap.
- Modifying (or updating) a heap object (or its components) is represented as $H[\ p \rightsquigarrow object\ ]$ on the left-hand side and as $H[\ p \rightsquigarrow updated\_object\ ]$ on the right-hand side of a rule.

As a matter of convenience that reduces the number of rules, we assume that each piece of code begins with a LAM n instruction, meaning that the respective spine is preceded by a sequence of n leading Λs, with $n = 0$ as a special case.

The state transition rules are enumerated as in Fig. 6.14 for the HOR machine to expose their close relationship.

The most complex of the instructions is LAM n which does the first part of a β-reduction. When executed from F, it has to distinguish between pushing a suspension pointer found in B as another entry onto stack S (rule (2a)), creating a new environment frame from entries held in S (rule (2b)), setting up in an empty structure B a LAM 0 instruction (rule (3a)), and lifting unapplied Λs to the leading Λ-cell that, in this particular state, is the sole entry in B (rule (3b)), thereby also incrementing the unapplied lambdas counts and pushing them onto stack S. Thus, all four rules combined in fact turn an *apps–lambs* corner into an environment frame. Rules (2a) and (2b) fill it with suspension entries and rules (3a) and (3b) do the equivalent of an η-extension by filling, in the case of a partial application, the entries for unapplied Λs with ULCs.

The second part of a β-reduction involves occurrences of the instruction VAR i. Depending on the parameter i, it either sets up for evaluation in the head a suspension retrieved from the environment (rule (5))[3] or it simply updates this parameter by the difference between the current ULC and a ULC entry found in the environment, signifying arrival at a head normal form, and reverses direction to execute backward code out of B (rule (6)).

Executing the instruction AP p from F creates a suspension for whatever object the parameter p refers to, and subsequently moves itself over into B with the suspension pointer as a parameter (rule (1)).

The three rules left in Fig. 8.1 apply to backward interpretation out of B of the instructions AP and LAM.

The instruction AP p_{sus} expects as its parameter the pointer to a suspension that it sets up for full normalization by a fresh machine (rule (9)). In the dump it saves a tuple headed by a symbol t, signifying reduction in a tail. The tuple also includes the suspension pointer as an anchor point and the structures F and B of the calling machine that must be restored when returning.

The instruction LAM n in conjunction with a nonempty dump marked t complements the backward execution of AP p_{sus}. It overwrites the suspension node with the normalized code and returns control to the calling machine, which creates in F another AP instruction, with the pointer to this code as a parameter.

A configuration with a LAM n instruction popping to the top of B and an empty dump terminates the machine with the code of a head-normalized spine in F. This code typically includes pointers that recursively refer to other pieces of head-normalized code held in the heap. Taken together, they form the code equivalent of a fully normalized Λ-expression.

[3] It is important to note that the ULC value u is taken over as in the current machine state, not as in the suspension, since the code of the suspension is directly inserted for VAR i, thus simply extending the current spine.

When comparing these rules with those of the HOR machine, it may be noted that there are some minor differences that are primarily related to code execution versus the transformation of expressions, but are also related to the representation of the environment. Collecting environment entries in the stack S before creating a frame and the particularities of handling a leading LAM instruction render it necessary to split HOR rules (2) and (3) up into two each. There is no explicit equivalent for HOR rule (4) as selecting an environment entry is taken care of, as part of rules (5) and (6), by the function *lookup*, and there is also no equivalent for HOR rule (8) since entering into and returning from the evaluation of tail suspensions is handled by saving and unsaving relevant machine states, including anchor points, in (from) the dump.

8.2.2 Interpretation Under Sharing in the Head **

Interpreting the B-machine instructions becomes more complicated, requiring more rules and also additional instructions, when the reduction of suspensions in the head is shared, as outlined in Sect. 7.2. The difficulties primarily concern the instruction VAR i. They are mainly caused by the fact that under sharing a suspension may be reduced to head normal form only in order to avoid potential termination problems. When abstractions thus head-normalized are applied in other contexts, suspensions of suspensions are, as an unpleasant consequence, created in their tails. Normalizing these nested suspensions is decidedly more involved than that of plain suspensions. The instruction must also distinguish between retrieving from the environment a suspension (of a suspension) and code that, due to prior evaluation under sharing, is already head-normalized and can therefore be directly executed.

Accordingly, the interpretation of VAR i must be split up into four different rules (5a–d) as given in Fig. 8.2, which replace rule (5) of Fig. 8.1 for the basic (nonsharing) B-machine. When retrieving, per lookup in the environment, a pointer *pp*, the instruction replaces itself by another instruction RTHNF *pp* that forces the reduction to head normal form of whatever *pp* refers to (rule (5a)). This instruction is not visible at the machine language level that is accessible to the user or the (de-)compiler. It just enters the scene and disappears again in the course of executing code, but it occurs neither in the initial nor in the terminal (normalized) code.

The state transition rules for the interpretation of RTHNF *pp* are the following: If *pp* points to

- head-normalized code, denoted as $hncode[e]$, it sets this code up for execution in the structure F (rule (5b));
- a suspension whose tail pointer refers to code, then it creates a fresh machine to execute this code in the environment and with the ULC that comes with the suspension, saving in the dump a return continuation earmarked h in its first component (for reduction in the head) (rule (5d));
- a suspension of a suspension, it again creates a fresh machine that recursively applies the instruction RTHNF to its tail pointer, thus driving the

(5a) $(\ (p_E \mid u),\ \text{VAR}\ i : nil,\ B,\ S,\ H,\ D,\ fw\ ) \mid (\ lookup(i,\ p_E) = pp\ )\ \rightarrow$
$(\ (p_E \mid u),\ \text{RTHNF}\ pp : nil,\ B,\ S,\ H[],\ D,\ fw\ )$

(5b) $(\ (p_E \mid u),\ \text{RTHNF}\ pp : nil,\ B,\ S,\ H[\ pp \rightsquigarrow hncode[e]\ ],\ D,\ fw\ )\ \rightarrow$
$(\ (p_E \mid u),\ hncode[e] : nil,\ B,\ S,\ H[],\ D,\ fw\ )$

(5c) $(\ (p_E \mid u),\ \text{RTHNF}\ pp : nil,\ B,\ S,$
$H[\ pp \rightsquigarrow (sus,\ (p'_E \mid u'),\ p'_t),\ p'_t \rightsquigarrow (sus, \ldots)\ ],\ D,\ fw\ )\ \rightarrow$
$(\ (p'_E \mid u'),\ \text{RTHNF}\ p'_t : nil,\ nil,\ S,\ H[],$
$(\ h,\ u,\ pp,\ \text{RTHNF}\ pp : nil,\ B,\ D\ ),\ fw\ )$

(5d) $(\ (p_E \mid u),\ \text{RTHNF}\ pp : nil,\ B,\ S,$
$H[\ pp \rightsquigarrow (sus,\ (p'_E \mid u'),\ p'_t),\ p'_t \rightsquigarrow code[e']\ ],\ D,\ fw\ )\ \rightarrow$
$(\ (p'_E \mid u'),\ code[e'] : nil,\ nil,\ S,\ H[],$
$(\ h,\ u,\ \ pp,\ \text{RTHNF}\ pp : nil,\ B,\ D\ ),\ fw\ )$

(8a) $(\ -,\ hncode[e'] : nil,\ \text{LAM}\ n : nil,\ S,\ H[\ pp \rightsquigarrow (sus, \ldots),\ p_{E0} \rightsquigarrow nil\ ],$
$(\ h,\ uu,\ pp,\ \text{RTHNF}\ pp : nil,\ B,\ D\ ),\ bw\ )\ \rightarrow$
$(\ (p_{ER},\ u),\ \text{RTHNF}\ pp : nil,\ B,\ S,$
$p_{ER} \rightsquigarrow <\ u,\ u,\ p_{E0} \mid u\ \ldots\ 1\ >: H[\ pp \rightsquigarrow \text{LAM}\ n : hncode[e'], \ldots\ ],\ D,\ fw\ )$

(8b) $(\ -,\ F,\ \text{AP}\ pp : B,\ S,\ H,\ (\ h, \ldots,\ D\ ),\ bw\ )\ \rightarrow$
$(\ -,\ \text{AP}\ pp : F,\ B,\ S,\ H, (\ h, \ldots,\ D\ ),\ bw\ )$

Fig. 8.2. The state transition rules for the interpretation of B- machine instructions under sharing

demand for the evaluation of nested suspensions recursively from outermost to innermost (rule (5c)).

The return continuations save the current ULCs and the instructions RTHNF pp so that, when code execution returns to the calling machines, the head-normalized code can be executed in place and in the correct context.

Returning to a calling machine is accomplished by rule (8a) of Fig. 8.2 that in the head does something similar to rule (7) of Fig. 8.1, which in the basic B-machine handles returning from the normalization of tail suspensions. It applies, under the backward execution mode, to a configuration that again features an instruction LAM n on top of an otherwise empty structure B, this time in conjunction with some piece of code in F that is just head-normalized. The rule prepends LAM n to the code in F, overwrites with it the suspension node whose pointer pp has been saved in the dump, then restores the calling machine from the dump and switches to the forward execution mode.

Restoring the calling machine includes the old codes in F and B and the construction of a new environment consisting of just a single frame that is filled in ascending order with as many ULCs as specified by the ULC value retrieved from the dump. This new environment is the one in which the head-normalized code must be executed in this particular place of call, after the

original environment of the suspension has served its purpose and is done with. This simply reflects the fact that this place has become the head of a straightened-out spine with a leading *lambs* sequence of length ULC. All accesses to this environment or, more precisely, to its single nonempty frame of ULCs by instructions VAR i that occur somewhere in the head-normalized code become turning points of code execution from forward to backward, signifying arrival at a head normal form.

The configuration returned by rule (8a) immediately matches the left-hand side of rule (5b). It causes the instruction RTHNF pp to set up the code referenced by pp for execution in this new environment. As long as there are nested return continuations in the dump that are earmarked h, the cycle of going through rules (8a) and then (5b) repeats itself, terminating either with an empty dump or, when recursively head-normalizing the spines of tails, with return continuations in the dump that are earmarked t.

Finally, the new rule (8b) of Fig. 8.2 applies to AP pp instructions in the code held in B. When executed in the backward mode under the control of a head-normalizing fresh machine (which can be identified by a return continuation tagged h in the dump), it is simply moved over into F without its parameter being touched.

Sharing reduction to full normal forms of suspensions in the tails, if it can be exploited at all, is implicitly taken care of by rules (9) and (7) of the basic B-machine. Here too, fresh machines are created to do the job, and the codes of the normalized suspensions overwrite the original suspension nodes so that they may be shared with multiple pointer occurrences. Copies of the suspension pointers that, upon returning to the calling machines, are retrieved from the dump subsequently also become the parameters of AP instructions set up in F to place them properly in the tails of apply nodes.

8.3 Executing B-Machine Code: an Example $**$

With precise specifications of the instructions at hand, we are now ready to go step by step through a small piece of program that illustrates how the B-machine does its job.

As an example, we consider the spine of a fairly simple head form with as yet unspecified tail expressions

$$\Lambda\Lambda \mathbin{@@} \Lambda\Lambda\Lambda \mathbin{@} \#i\ e_1\ e_2\ e_3$$

(with $i \in \{0, \ldots, 4\}$), which is graphically depicted in Fig. 8.3. It translates straightforwardly into the B-machine code

$$\text{LAM } 2;\ \text{AP } p_3;\ \text{AP } p_2;\ \text{LAM } 3;\ \text{AP } p_1;\ \text{VAR } i;$$

that must be set up in F and executed from left to right. The codes that compute the tail expressions $e_1,\ e_2,\ e_3$ are referenced by the pointer parameters $p_1,\ p_2,\ p_3$, respectively, of the instructions AP.

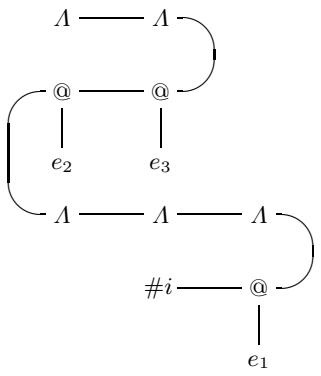


Fig. 8.3. An example graph

Figure 8.4 illustrates some characteristic phases of executing this piece of code from F. It also shows the code that builds up in B, the contents of the workspace stack S and of the environment section H_E of the heap as code execution proceeds. The currently valid environment pointer is the one that in H_E is underlined. The downward-pointing arrows have attached to them the current ULCs and the rules that effect the state changes, as enumerated in Fig. 8.1.

Beginning with the complete code in F, with the ULC set to zero and all other structures empty, and moving forward, the first instruction LAM 2 of F, finding B empty, creates, through a sequence of five rule applications, an environment frame containing two ULCs in ascending order that it prepends to the empty environment (fourth configuration from the top). The two AP instructions that follow next in F then move themselves over into B with parameters pp_i that are pointers to suspensions created by pairing the pointers p_i to the tail codes with the environment pointer p_{E1} (fifth configuration from the top). Then the instruction LAM 3 takes over. It removes two AP instructions from B before hitting the LAM 2 instruction underneath, pushes their suspension pointers onto S and decrements its own arity index to one. The ensuing combination of LAM 1 in F and LAM 2 in B pushes the ULC value 3 onto S, and changes the parameters of these instructions to 0 and 3, respectively (second last configuration), which leads to the creation of another environment frame of three entries. Subsequently moving the instruction AP p_1 that now is next in F over into B yields the last configuration of Fig. 8.4. It has the VAR i instruction exposed in F as the last instruction of this particular piece of code, whose index selects, through the function $lookup(i,\ p_{E2})$, an entry in the current environment.

If the head index is set to $i \in \{0,\ 3,\ 4\}$, which means that it is bound by the leading *lambs* sequence, the machine has to go into reverse gear to

$$\begin{array}{r}
\textsc{lam}\ 2 : \textsc{ap}\ p_3 : \textsc{ap}\ p_2 : \textsc{lam}\ 3 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
nil \mid B \\
nil \mid S \\
\underline{p_{E0}} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 0 \mid$ rule (3a)

$$\begin{array}{r}
\textsc{lam}\ 2 : \textsc{ap}\ p_3 : \textsc{ap}\ p_2 : \textsc{lam}\ 3 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
\textsc{lam}\ 0\ :\ nil \mid B \\
nil \mid S \\
\underline{p_{E0}} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 0 \mid$ rule (3b) twice (fw)

$$\begin{array}{r}
\textsc{lam}\ 0 : \textsc{ap}\ p_3 : \textsc{ap}\ p_2 : \textsc{lam}\ 3 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
\textsc{lam}\ 2\ :\ nil \mid B \\
2 : 1 : nil \mid S \\
\underline{p_{E0}} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 2 \mid$ rule (2b) twice (fw)

$$\begin{array}{r}
\textsc{ap}\ p_3 : \textsc{ap}\ p_2 : \textsc{lam}\ 3 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
\textsc{lam}\ 2\ :\ nil \mid B \\
nil \mid S \\
\underline{p_{E1}} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 2 \mid$ rule (1) twice (fw)

$$\begin{array}{r}
\textsc{lam}\ 3 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
\textsc{ap}\ pp_2 : \textsc{ap}\ pp_3 : \textsc{lam}\ 2\ :\ nil \mid B \\
nil \mid S \\
\underline{p_{E1}} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 3 \mid$ rule (2a) twice and rule (3b) (fw)

$$\begin{array}{r}
\textsc{lam}\ 0 : \textsc{ap}\ p_1 : \textsc{var}\ i\ :\ nil \mid F \\
\textsc{lam}\ 3\ :\ nil \mid B \\
3 : pp_3 : pp_2 : nil \mid S \\
\underline{p_{E1}} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

$\Downarrow\ u = 3 \mid$ rules (2b) and (1) (fw)

$$\begin{array}{r}
\textsc{var}\ i\ :\ nil \mid F \\
\textsc{ap}\ p_1 : \textsc{lam}\ 3\ :\ nil \mid B \\
nil \mid S \\
\underline{p_{E2}} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

Fig. 8.4. Phases of forward code execution in the B-machine

generate, by execution of the code in B, the code equivalent of a normalized spine in F.

If the head index assumes one of the values $i \in \{1, 2\}$, then VAR i in fact becomes a branch instruction that continues executing from F the code of the suspension referenced by either pp_2 or pp_3, both in the environment p_{E2}.

In order to discuss what happens in the latter case, we assume a head index of $i = 2$ and choose for the tail expressions the following codes:

$$\begin{array}{ll} p_1 \rightsquigarrow \text{LAM } 1; \text{ VAR } 2; & \text{for the tail } e_1 \\ p_2 \rightsquigarrow \text{LAM } 1; \text{ VAR } 0; & \text{for the tail } e_2 \\ p_3 \rightsquigarrow \text{LAM } 1; \text{ VAR } 1; & \text{for the tail } e_3\ . \end{array}$$

These codes bring out the essentials of continuing with reductions in the head but are simple enough to keep the number of rule applications reasonably small.

The complete code thus realizes the Λ-expression

$$\Lambda\Lambda\ @\ @\ \Lambda\Lambda\Lambda\ @\ \#2\ \Lambda\#2\ \ \Lambda\#0\ \ \Lambda\#1$$

or, in parenthesized notation,

$$\Lambda\Lambda.((\Lambda\Lambda\Lambda.(\#2\ \Lambda\#2)\ \Lambda\#0)\ \Lambda\#1)\ \ .$$

As may be easily verified, this expression reduces in three β-reduction steps to

$$\Lambda\Lambda\Lambda\Lambda\Lambda\#3\quad .$$

Figure 8.5 depicts as a sequence of machine configurations how these reductions can be performed by the above codes, using first the instruction interpretation specified by the rules of Fig. 8.1 only, i.e., without going through the motions that would be necessary to do sharing in the head.

The sequence begins with the last configuration of Fig. 8.4, which has the instruction VAR 2 as the last one left in F.[4] It selects from the current environment p_{E2} the suspension referenced by pp_2 and sets it up for evaluation in the head (second configuration from the top). These steps are repeated in the fourth and sixth configurations from the top for the suspensions referenced by $pp1$ and $pp3$, respectively. The interesting case with regard to index corrections comes about in the second last configuration. Here we have exposed in F an instruction VAR 1 in conjunction with a ULC value of 5 and an environment composed of two frames. The index $i = 1$ now selects the first ULC entry 2 of the second frame (the one to which p_{E1} is pointing). The difference of 3 between the two ULCs becomes the updated parameter of the instruction VAR. In the last step, the machine prepends the instruction LAM 5 in B to it to form in F the code equivalent

[4] Note that stack S has been dropped throughout the entire sequence since it is not directly involved, and that the heap section H_S that holds the suspensions and the dump are shown only once in the first configuration since they never change.

$$\begin{array}{r}
\textsc{var}\ 2 : nil \mid F \\
\textsc{ap}\ pp_1 : \textsc{lam}\ 3\ :\ nil \mid B \\
\underline{p_{E2}} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
pp_2 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_2),\ pp_3 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_3) \mid H_S \\
pp_1 \rightsquigarrow (sus,\ (p_{E2} \mid 3),\ p_1) \mid H_S \\
nil \mid D
\end{array}$$

$$\Downarrow\ \ u = 3 \mid \text{rule (5) } (fw)$$

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 0 : nil \mid F \\
\textsc{ap}\ pp_1 : \textsc{lam}\ 3\ :\ nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ \underline{p_{E1}} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 3 \mid \text{rules (2a) and (2b) } (fw)$$

$$\begin{array}{r}
\textsc{var}\ 0 : nil \mid F \\
\textsc{lam}\ 3 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE1}} \rightsquigarrow <\ 1,\ 2,\ p_{E1} \mid pp_1\ > \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 3 \mid \text{rule (5) } (fw)$$

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 2 : nil \mid F \\
\textsc{lam}\ 3 : nil \mid B \\
\underline{p_{E2}} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
p_{EE1} \rightsquigarrow <\ 1,\ 2,\ p_{E1} \mid pp_1\ > \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 4 \mid \text{rule (3b) } (fw)$$

$$\begin{array}{r}
\textsc{var}\ 2 : nil \mid F \\
\textsc{lam}\ 4 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE2}} \rightsquigarrow <\ 1,\ 4,\ p_{E2} \mid 4\ >, p_{EE1} \rightsquigarrow <\ 1,\ 2,\ p_{E1} \mid pp_1\ > \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 4 \mid \text{rule (5) } (fw)$$

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 1 : nil \mid F \\
\textsc{lam}\ 4 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ \underline{p_{E1}} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE2}} \rightsquigarrow <\ 1,\ 4,\ p_{E2} \mid 4\ >, p_{EE1} \rightsquigarrow <\ 1,\ 2,\ p_{E1} \mid pp_1\ > \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 5 \mid \text{rule (3b) } (fw)$$

$$\begin{array}{r}
\textsc{var}\ 1 : nil \mid F \\
\textsc{lam}\ 5 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >,\ p_{E1} \rightsquigarrow <\ 2,\ 2,\ p_{E0} \mid 2\ 1\ >,\ p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE3}} \rightsquigarrow <\ 1,\ 5,\ p_{E1} \mid 5\ >,\ p_{EE2} \rightsquigarrow \ldots,\ p_{EE1} \rightsquigarrow \ldots \mid H_E
\end{array}$$

$$\Downarrow\ \ u = 5 \mid \text{rule (6) } (fw)$$

$$\begin{array}{r}
\textsc{lam}\ 5 : \textsc{var}\ 3 : nil \mid F \\
nil \mid B \\
p_{EE3} \rightsquigarrow \ldots,\ p_{EE2} \rightsquigarrow \ldots,\ p_{EE1} \rightsquigarrow \ldots,\ p_{E2} \rightsquigarrow \ldots,\ p_{E1} \rightsquigarrow \ldots,\ p_{E0} \rightsquigarrow nil \mid H_E
\end{array}$$

Fig. 8.5. Continuing with nonshared reductions in the head

$$\text{LAM } 5 : \text{VAR } 3 : nil$$

of the normalized expression.

Though this example cannot really exploit any sharing, we can at least show how code-controlled reductions in the head can be performed in the isolation of a fresh machine and how the computation continues with the code returned by it.

Figure 8.6 depicts the first part of the sequence of state transformations that develops when, beginning at the same configuration as in Fig. 8.5, the instructions are interpreted under sharing, using whenever applicable the rules listed in Fig. 8.2, and otherwise the rules of Fig. 8.1.

The first step sees rules (5a) and (5d) in action. They have the instruction VAR 2 in F select from the current environment the suspension referenced by pp_2 and set it up for execution in a fresh machine by saving the current machine state in the dump (second configuration from the top). This machine can do nothing but overwrite the suspension with its abstraction code as it is, prepend to the current environment a frame of ULCs that derive from the ULC saved in the dump, and then return control to top level, where the abstraction code is set up for execution out of F (fourth configuration from the top).

The VAR 0 instruction that then pops to the top of F (second last configuration) selects the suspension referenced by pp_1 and, by calling the instruction RTHNF pp_2, creates another fresh machine for its shared head normalization (last configuration). The computation would then continue by recursively calling inside the machine for pp_1 yet another machine to reduce the suspension referenced by pp_3 and, after returning from both machines, produce the code LAM 5 :VAR 3 as the full normal form.

The point to be driven home here is that it takes two passes through the code to first head-normalize a suspension for sharing and then to apply this code in the head of the spine in which it is called first. All subsequent calls, of which there are none in our trivial example, may use the head-normalized code directly. A more complex suspension would go through essentially the same motions of calling and returning from a fresh machine; otherwise the machine would just perform more computational steps.

8.4 Supporting Primitive Functions

Primitive functions such as $+$, $-$, ... are strict in their arguments and thus are a little at odds with the head-order regime. The simpler case seems to be that a primitive function symbol can be statically inferred to be in the head of a spine, e.g., as in @ @ $+$ e_1 e_2. It could be translated into something like

$$code[e_2]; code[e_1]; \text{ ADD}; \quad .$$

When executed from left to right, this piece of code would first evaluate the arguments e_2 and e_1 in that order, and then add them up. We may assume

$$\begin{array}{r}
\textsc{var}\ 2 : nil \mid F \\
\textsc{ap}\ pp_1 : \textsc{lam}\ 3\ :\ nil \mid B \\
\underline{p_{E2}} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
pp_2 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_2),\ pp_3 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_3) \mid H_S \\
pp_1 \rightsquigarrow (sus,\ (p_{E2} \mid 3),\ p_1) \mid H_S \\
p_1 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 2 : nil,\ p_2 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 0 : nil,\ p_3 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 1 : nil \mid H_S \\
nil \mid D
\end{array}$$

$\Downarrow$ $u = 3$ | rules (5a) and (5d) (fw)

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 0 : nil \mid F \\
nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, \underline{p_{E1}} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
\ldots,\ pp_2 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_2), p_2 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 0 : nil, \ldots \mid H_S \\
(\ h,\ 3,\ pp_2, \textsc{rthnf}\ pp_2 : nil, \textsc{ap}\ pp_1 : \textsc{lam}\ 3 : nil,\ nil\ ) \mid D
\end{array}$$

$\Downarrow$ $u = 2$ | rules (3a), (3b), (2b) and (6) (fw)

$$\begin{array}{r}
\textsc{var}\ 0 : nil \mid F \\
\textsc{lam}\ 1 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE1}} \rightsquigarrow <\ 1,\ 3,\ p_{E1} \mid 3 > \mid H_E \\
\ldots,\ pp_2 \rightsquigarrow (sus,\ (p_{E1} \mid 2),\ p_2), p_2 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 0 : nil, \ldots \mid H_S \\
(\ h,\ 3,\ pp_2, \textsc{rthnf}\ pp_2 : nil, \textsc{ap}\ pp_1 : \textsc{lam}\ 3 : nil,\ nil\ ) \mid D
\end{array}$$

$\Downarrow$ $u = 3$ | rules (8a) (bw) and (5b) (fw)

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 0 : nil \mid F \\
\textsc{ap}\ pp_1 : \textsc{lam}\ 3 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{ER1}} \rightsquigarrow <\ 3,\ 3,\ p_{E0} \mid 3\ 2\ 1 >, p_{EE1} \rightsquigarrow <\ 1,\ 3,\ p_{E1} \mid 3 > \mid H_E \\
\ldots,\ pp_2 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 0 : nil, p_2 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 0 : nil, \ldots \mid H_S \\
nil \mid D
\end{array}$$

$\Downarrow$ $u = 3$ | rules (2a) and (2b) (fw)

$$\begin{array}{r}
\textsc{var}\ 0 : nil \mid F \\
\textsc{lam}\ 3 : nil \mid B \\
p_{E2} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
\underline{p_{EE2}} \rightsquigarrow <\ 1,\ 3,\ p_{ER1} \mid pp_1 >,\ p_{ER1} \rightsquigarrow <\ 3,\ 3,\ p_{E0} \mid 3\ 2\ 1 >,\ p_{EE1} \rightsquigarrow \ldots \mid H_E \\
\ldots,\ pp_1 \rightsquigarrow (sus,\ (p_{E2} \mid 3),\ p_1), p_1 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 2 : nil, \ldots \mid H_S \\
nil \mid D
\end{array}$$

$\Downarrow$ $u = 3$ | rules (5a) and (5d) (fw)

$$\begin{array}{r}
\textsc{lam}\ 1 : \textsc{var}\ 2 : nil \mid F \\
nil \mid B \\
\underline{p_{E2}} \rightsquigarrow <\ 3,\ 3,\ p_{E1} \mid 3\ pp_3\ pp_2\ >, p_{E1} \rightsquigarrow < 2,\ 2,\ p_{E0} \mid 2\ 1 >, p_{E0} \rightsquigarrow nil \mid H_E \\
p_{EE2} \rightsquigarrow <\ 1,\ 3,\ p_{ER1} \mid pp_1 >,\ p_{ER1} \rightsquigarrow <\ 3,\ 3,\ p_{E0} \mid 3\ 2\ 1 >,\ p_{EE1} \rightsquigarrow \ldots \mid H_E \\
\ldots,\ pp_1 \rightsquigarrow (sus,\ (p_{E2} \mid 3),\ p_1), p_1 \rightsquigarrow \textsc{lam}\ 1 : \textsc{var}\ 2 : nil, \ldots \mid H_S \\
(\ h,\ 3,\ pp_1, \textsc{rthnf}\ pp_1 : nil, \textsc{lam}\ 3 : nil,\ nil\ ) \mid D
\end{array}$$

Fig. 8.6. Continuing with shared reductions in the head

here that the argument values would be temporarily pushed by the respective codes onto stack S, from where they could be picked up by ADD and replaced with the result value.

Unfortunately, this solution does not take into account the general case where we may have in the head positions a variable (or a binding index, for that matter) that, in the course of performing a code-controlled β-reduction, may be substituted by a primitive function.

A workaround that is compatible with the β-reduction mechanics of the B-machine consists in translating occurrences of primitive function symbols such as + into the abstraction code

$$p_{add} \rightsquigarrow \text{LAM } 2;\ \text{FORCE } 0\ ;\ \text{FORCE } 1;\ \text{ADD; FREE;} \quad .$$

As the name implies, the instruction FORCE forces evaluation in the current environment of the suspension selected by its index parameter, whose result value is then deposited in stack S for subsequent consumption by the ADD instruction. The instruction FREE releases the environment frame created by the instruction LAM 2 since it is used nowhere else. Using the pointer p_{add} as a handle, this piece of code can be freely passed around by means of β-reductions and called for execution wherever it ends up in the head of a spine.

Similar codes may be given for all other primitive functions, including conditionals, though we seem to face a minor difficulty here insofar as alternative codes must be executed depending on the outcome of evaluating a predicate expression. However, the Λ-calculus takes care of this by representing the Boolean values `true` and `false` as selector abstractions $\Lambda\Lambda\ \#1$ and $\Lambda\Lambda\ \#0$, respectively, which translate into the codes

$$p_{true} \rightsquigarrow \text{LAM } 2;\ \text{VAR } 1; \quad \text{and} \quad p_{false} \rightsquigarrow \text{LAM } 2;\ \text{VAR } 0; \quad .$$

Thus, all we need to do is to define the code for a primitive function `if` as

$$p_{if} \rightsquigarrow \text{LAM } 1;\ \text{FORCE } 0;\ \text{BRANCH;} \quad .$$

It takes a single argument that is assumed to be a predicate expression, forces its evaluation and, depending on the Boolean value, deposits either the pointer p_{true} or p_{false} in stack S. This pointer is taken by the instruction BRANCH to continue execution in the head with the code referenced by it.

When applying the function `if` as

$$@\ @\ @\ \texttt{if}\ e_0\ e_1\ e_2 \quad ,$$

the equivalent code looks like this:

$$\text{AP } p_2;\ \text{AP } p_1;\ \text{AP } p_0;\ \text{BSR } p_{if};$$

(where p_0, p_1, p_2 point to the codes $code[e_0]$, $code[e_1]$, $code[e_2]$, respectively, and BSR denotes another simple branch instruction). This code first generates AP instructions for all three arguments held in B and then calls the code for `if` to evaluate e_0 in its environment. This code in turn calls the code referenced

by either p_{true} or p_{false} and thus decides whether to continue with execution of the code for e_1 or e_2 out of F.

We note that primitive functions translated into such codes in fact realize closed Λ-abstractions, or combinators, that may be called without regard for the current environment.

The combinator property also renders it possible to include the codes for all primitive functions in a runtime library that may be linked to executable B-machine code.

8.5 Summary

The original B-machine was designed by Berkling as a pure Λ-calculus machine with hardware realization in mind. The idea for this machine was born out of some frustration about difficulties in cleanly and efficiently resolving naming conflicts among all-quantified variables in the context of mechanized Horn-clause resolution. The solution to this problem was thought to be a novel machine architecture that supports a full-fledged Λ-calculus, with a complete and efficient implementation of the β-reduction rule as the single most important operation.

The starting point was the head-order reduction concept outlined in Sect. 6.4. It led to the idea of doing β-reductions in the large, i.e., of reducing several consecutive β-redices in one conceptual step, and of consequently focusing on reductions in the head of a spine until there was nothing else to do but to turn (recursively) to leftover tails. This variant of a normal order regime, in conjunction with sharing in the head, ensures that all β-reductions are performed at most once and only if absolutely necessary to compute normal forms.

Beyond that, it was clear from the beginning that raw runtime efficiency could be achieved only by instruction-based machinery rather than by interpretation of machine states. As a truly novel concept that directly derives from the head-order regime of going up and down the spine of a graph, the B-machine supports two instruction streams, of which one is executed in forward direction and, while this is being done, dynamically generates the other stream that must be executed backward. The forward stream (or code) is statically generated by compilation of the Λ-expression to be reduced and corresponds to reducing a spine of the expression graph to head normal form (or just going down the spine). The backward executing code controls the reduction of the head-normalized spine's tails (which recursively calls upon forward code execution) and corresponds to going up again along a spine. This code in turn generates a normalized spine as forward code.

Forward code is composed of three instructions, of which AP essentially constructs suspensions for tail expressions, LAM controls the creation of environment frames and VAR handles the head index of a spine. The latter instruction either returns an updated binding index and subsequently turns control

over to the backward code, or it continues with the evaluation in forward direction of a suspension selected from the environment.

The most important backward executing instruction is AP. It effects the evaluation of suspensions for tail expressions by forward code.

Reducing suspensions in the head or in a tail is separated from the rest of the spine by creating fresh machines (or contexts) that render it possible to share the (head) normal forms they compute with references to the suspensions somewhere else in the code.

Unfortunately, things are not as nice and tidy as they may look on the B-machine code level. The difficulties are largely hidden behind the two forward instructions LAM and VAR i that play a key role in performing β-reductions. Since each requires interpretation of different machine states, their execution turns out to be fairly complex, though some of the complexity can be avoided by introducing more specialized instructions. It may then be left to compilation to decide, based on a static program analysis, where these specialized instructions can safely replace those that cover the general cases (or all relevant machine configurations).

However, a complexity problem that is hard to overcome is due to sharing in the head. It relates to the question of how far reducing a suspension in the isolation of a fresh machine should be driven ahead, assuming no advance knowledge of what its (head) normal form might be. Since it could be an abstraction, there are basically three options available.

At one extreme, we could take the coward's approach of reducing the abstraction to weak normal form only, in which case there would be little benefit from sharing since it would fail to utilize opportunities for optimizing the abstraction itself. At the other extreme, we could go for full normalization, but here we face the problem that uninstantiated parameters may inflict runaway recursions in tail expressions that would otherwise be discarded, i.e., it does not guarantee termination, even though normal forms may exist.

Reducing to head normal forms only is the best solution conceptually, as it avoids these termination problems and also allows some optimizations under abstractions, but it can only be had at the expense of creating at runtime suspensions of suspensions that could extend over several nesting levels. Applying in different contexts abstractions thus head-normalized is bound to inflict a substantial overhead as these nestings have to be recursively reduced from the inside out, meaning that the machine has to run the particular code as many times as there are nestings. In worst cases, this overhead may easily offset what has been gained, in terms of runtime efficiency, by saving a number of β-reductions that would otherwise have to be performed repeatedly.

References

This chapter is largely based on unpublished paper drafts and handwritten notes that were privately communicated to the author by Berkling [Ber96, Ber97], and on the PhD theses by Hilton and Troullinos [Hil90, Trou93].

Related work on fully normalizing λ-calculus machines is rather scarce. Crégut describes in [Cre90] an abstract machine for full normalization of λ-terms. It is based on Krivine's weakly normalizing $\mathcal{K}$-machine that recursively reduces closures, using an updating scheme for binding indices that, owing to the nature of the problem, is similar to that for the ULCs of the B-machine. Another implementation of full normalization is due to Grégoire and Leroy [GrLe02]. It too uses a weakly normalizing machine, augmented by a so-called readback function, which α-converts λ-bound variables into fresh variables or binding indices, just as required by the classical definition of β-reduction (see Sect. 4.2), before normalizing abstraction bodies. This readback function is also recursively applied to the arguments of applications of free variables.

Yet another approach, first published in [GK96] and extensively treated in Chap. 10 of this text, again uses weakly normalizing code-executing machinery to do the routine work of naive β-reductions but switches to an η-extension mechanism that turns unapplied or partially applied abstractions into full applications to recursively enable further weak reductions in abstraction bodies.

9

The *G*-Machine

The preceding chapter showed that a fully normalizing λ-calculus does not lend itself smoothly to code execution as it is known from conventional computing machines. Transforming abstractions into others, i.e., performing reductions in abstraction bodies, which is what distinguishes full from weak normalization, requires ways and means of modifying code dynamically. To do so, the *B*-machine supports two instruction streams, of which the one obtained by compilation of λ-expressions and executed in what is called the forward direction, dynamically generates the other one that is carried out in the reverse direction to generate code that represents a fully normalized expression. Instruction execution also involves a considerable amount of interpretation of machine states to do the right things in the right order and, most importantly, to maintain correct variable bindings.

Contemporary computing machines are ill equipped for this purpose. Executable code is expected to be static, there is no direct way of dynamically generating new from existing code (other than by recompilation), and there are no instructions that are a reasonably close match to those of the *B*-machine.

Conventional instruction sets, loosely speaking, expect all data objects that they are supposed to operate on to be of the right type and format in the right places (memory locations) at the right times (or states of control). Other than for conditional branch instructions that inspect a few status bits, instruction execution does not depend on the machine state. The machine blindly interprets as instructions the bit patterns to which the program counter is pointing, and correct code execution depends solely on this counter starting at the right address and being correctly advanced.

Of course, as anything that is Turing-computable can be implemented on such primitive target machinery, it is possible in principle to have *B*-machine code either interpreted by or compiled to target code, though this cannot be expected to be very efficient both in terms of time and memory space expended.

In order to generate efficient target code, some restrictions must inevitably be imposed on the high-level source language as regards the construction of

legitimate algorithms (or programs). These restrictions must be tight enough to rule out anything that at runtime leaves a choice between two or more alternative courses of action or, even worse, brings about undecidable situations.

Since the λ-calculus treats abstractions and variables as first-class objects, there is simply too much freedom in designing algorithms, including many that are not very meaningful semantically (see Chap. 2). This freedom renders it impossible for a compiler to statically infer in all cases whether and to what extent applications can actually be reduced with regard to compatible argument types, matching arities of applications and abstractions, and such fairly complicated things as the handling of potential naming conflicts.

However, it is perfectly reasonable to challenge the freedom of a full-fledged λ-calculus when it comes to writing algorithms that solve real-life application problems and to executing them efficiently. There are good reasons to argue that full support for functions and variables as first-class objects is a luxury that is rarely used, and that a rigorous typing discipline not only raises confidence in the correctness of algorithms but also considerably facilitates compilation to efficiently executable code.

The canonical approach taken in the world of functional languages is referred to as compiled graph reduction. It is based on a weakly normalizing λ-calculus and on constructing and modifying graph representations of λ-expressions that resemble those used by the interpreting graph reducer of Chap. 7 but, owing to the absence of full-fledged β-reductions, turns out to be decidedly simpler.

The idea is to compile the (nested) sets of function definitions of an algorithm written in some high-level language such as AL into code for some suitably defined abstract machine. This code starts out with the construction of a graph for the outermost goal expression of the algorithm and, in the course of reducing it, repeatedly constructs and subsequently reduces application graphs by calling function codes held in some persistent part of memory, until a graph in weak normal form is reached.

Weak normalization must be seen primarily as a consequence of compiling functions to code. Since codes are static objects that transform the graphs that they operate on but are not graphs themselves, we have in fact a separation of the world of functions from the world of those objects that are legitimate function arguments and function values. In particular, functions cannot be freely applied to other functions to compute new functions and use them as operators elsewhere, i.e., in a strict sense, functions lose their status as first-class objects. What can be done, though, is to pass function codes as parameters to other function codes.

Compilation to static code also requires that functions of n formal parameters be applied to n actual parameters (arguments), usually of specific types, for the code to execute correctly, which – strictly speaking – rules out partial applications.

The workaround in both cases is usually to wrap the components of the respective applications up in closures, and to pass them along in this form until they can be unwrapped again elsewhere once they have picked up the missing arguments, or otherwise must be returned in unintelligible form as (part of) the result.

The most prominent abstract machine for compiled graph reduction that for a long time has set a standard for the implementation of functional languages with a lazy semantics is the *G*-machine invented by Johnsson. It defines an intermediate level of code generation and code execution which assumes that high-level functional programs composed of nested sets of function definitions are, prior to compilation, converted into flat sets of supercombinators. The programs must also be well typed so that no type inconsistencies can occur at runtime.

Supercombinators offer some interesting conceptual advantages when it comes to compiling them to code of an abstract or real machine. They are defined to be closed abstractions that have all abstractions that may occur in their body expressions recursively closed (or turned into supercombinators) as well. With all variables bound, supercombinator reduction cannot run into naming conflicts, i.e., all substitutions may safely be performed naively. Supercombinators represented as closed abstractions of the nameless Λ-calculus, as we have seen in Sect. 4.3, have the pleasant property of not changing any binding indices when reducing full applications. A compiler may convert these indices into fixed offsets relative to the base of an environment that consists of just a single contiguous frame of argument entries.

In the following we will describe the workings of this machine, introduce a set of instructions that are essential for graph manipulations that relate to function calls, and specify the rules for compiling supercombinators to *G*-code, including some code optimizations.

9.1 Basic Language Issues

The *G*-compiler accepts as input program expressions specified as sets of mutually recursive function definitions of the form

$$\begin{aligned}&\texttt{define}\\ &\qquad f_1 \;=\; \lambda u_{11}\;\ldots\;u_{1n}.e_1\\ &\qquad\quad\ldots\\ &\qquad f_r \;=\; \lambda u_{r1}\;\ldots\;u_{rn}.e_r\\ &\texttt{in}\; e_0\;\;,\end{aligned}$$

as we know it from the language AL. However, as just indicated, there is an important difference that makes things a lot easier as far as the construction of the runtime environment and compilation to abstract machine code are concerned: the abstractions on the right-hand sides of the defining equations need to be all closed (or supercombinators), and so must be the goal expression

e_0. Without free variables, all function definitions may be given in flat form, or on the same level, since there can be no variable bindings that would require defining functions local to others.

The expressions $e_0,\ e_1, \ldots,\ e_r$ that are of primary interest here may be[1]

- variables, including identifiers of supercombinators, constant values and primitive function symbols;
- applications $(e'_0\ e'_1 \ldots e'_n)$ that are assumed to be curried, with association to the left, as usual.

The important point is that these expressions may contain neither anonymous abstractions nor other `define` constructs.

Algorithms written, say, in AL that may contain (relatively) free variables in functions that are defined to be local to others can be readily converted into supercombinator form. All it takes is to systematically abstract free variables out of open λ-abstractions to the next higher level of abstractors, and then lift all abstractions thus closed to the top level.

For an illustration of how this works, we consider a `def` construct that is assumed to be local to some other function definition:[2]

$$
\begin{array}{l}
\texttt{def} \\
\quad f = \lambda x.\ \dashv \ldots u \ldots g \ldots h \ldots v \ldots f \ldots \vdash \\
\quad g = \lambda y.\ \dashv \ldots g \ldots h \ldots w \ldots \vdash \\
\quad h = \lambda z.\ \dashv \ldots h \ldots g \ldots \vdash \\
\texttt{in} \quad \dashv \ldots h \ldots g \ldots f \ldots \vdash
\end{array}
$$

(again, the symbols $\dashv$ and $\vdash$ denote delimiters of the abstraction bodies). Here we have three mutually recursive functions $f,\ g,\ h$ in which we assume to have occurrences of free variables $u,\ v,\ w$ that may be bound somewhere higher up. To simplify this example a little, we also assume that the abstraction bodies do not contain further abstractions.

Abstracting (or lifting) free variables makes use of the semantic equivalence

$$e = (\lambda u.e\ u)\ \ ,$$

where u is assumed to be free in e. It may be used to transform, in a first step, the above set of abstractions into

$$
\begin{array}{l}
\texttt{def} \\
\quad f = \lambda u.\lambda v.\lambda x.\ \dashv \ldots u \ldots (g\ w) \ldots h \ldots v \ldots (f\ u\ v) \ldots \vdash \\
\quad g = \lambda w.\lambda y.\ \dashv \ldots (g\ w) \ldots h\ \ldots w \ldots \vdash \\
\quad h = \lambda z.\ \dashv \ldots h \ldots (g\ w) \ldots \vdash \\
\texttt{in} \quad \dashv \ldots h \ldots (g\ w) \ldots (f\ u\ v) \ldots \vdash \quad .
\end{array}
$$

[1] Legitimate expressions may also include local variable definitions (recursive and nonrecursive) which, however, will not be considered.

[2] The `def` construct differs from the `define` construct insofar as the functions defined under it need not be closed.

A close look at these new abstractions reveals that we are not yet done, as we have now introduced w as a new free variable in f and h. Repeating this λ-lifting step once more yields

$$\begin{array}{l}\texttt{define}\\ \quad f = \lambda w.\lambda u.\lambda v.\lambda x.\ \dashv \ldots u \ldots (g\ w) \ldots (h\ w) \ldots v \ldots (f\ w\ u\ v) \ldots \vdash\\ \quad g = \lambda w.\lambda y.\ \dashv \ldots (g\ w) \ldots (h\ w)\ \ldots w \ldots \vdash\\ \quad h = \lambda w.\lambda z.\ \dashv \ldots (h\ w) \ldots (g\ w) \ldots \vdash\\ \texttt{in}\quad \dashv \ldots (h\ w) \ldots (g\ w) \ldots (f\ w\ u\ v) \ldots \vdash\ ,\end{array}$$

in which all functions are now closed.[3]

Before compiling these supercombinators to G-machine code, all bound variables may be converted into binding indices for direct translation into environment offsets:

$$\begin{array}{l}\texttt{define}\\ \quad f = \Lambda\Lambda\Lambda\Lambda.\ \dashv \ldots \#2 \ldots (g\ \#3) \ldots (h\ \#3) \ldots \#1 \ldots (f\ \#3\ \#2\ \#1) \ldots \vdash\\ \quad g = \Lambda\Lambda.\ \dashv \ldots (g\ \#1) \ldots (h\ \#1) \ldots \#1 \ldots \vdash\\ \quad h = \Lambda\Lambda.\ \dashv \ldots (h\ \#1) \ldots (g\ \#1) \ldots \vdash\\ \texttt{in}\quad \dashv \ldots (h\ \#i_w) \ldots (g\ \#i_w) \ldots (f\ \#i_w\ \#i_u\ \#i_v) \ldots \vdash\ .\end{array}$$

We note that inside the abstraction bodies all indices have values smaller than the number of Λs that precede them, and that occurrences of the variables u, v, w in the goal expression are replaced by indices $\#i_u$, $\#i_v$, $\#i_w$, respectively, which denote their distances relative to the Λs that bind them in a larger context.

Once all nested sets of function definitions have thus been transformed into sets of supercombinators, they may be lifted to the same (top) level since all dependencies among them through variable (or index) occurrences that are bound higher up are then eliminated.

9.2 Basic Operating Principles of the *G*-Machine

The basic G-machine works with four runtime structures. These are

- a control structure C that holds the G-code to be executed;
- a runtime stack S for pointers to graph nodes that represent environments, in G-machine terminology also referred to as contexts, in which supercombinator reductions take place;
- a graph representation G of the expression that the code held in C is operating on;
- a dump stack D to handle supercombinator calls.

[3] The complexity of going repeatedly through function definitions to lift free variables may be avoided to some extent by a more elegant approach based on the solution of set equations. However, its worst-case complexity is still $O(k^3)$, with k being the number of function definitions involved.

There is also a persistent global structure that associates supercombinator names to codes. This structure is not included in the state description of the machine since it is of no relevance with regard to its dynamic behavior.

The instructions of the core G-machine may be defined by means of a state transition function

$$\tau_g : (S,\ G,\ C,\ D) \rightarrow (S',\ G',\ C',\ D') \quad .$$

To get an idea of just what instructions are needed and how they operate on the runtime structures, we first have to explain informally how the G-machine basically works. Not very surprisingly, it has a great deal in common with the head-order reducers of Chaps. 7 and 8.

The basic mechanism of the G-machine is designed to reduce supercombinator applications. It involves primarily the code, the graph and the stack. The code repeatedly constructs, from the contexts held in the stack, new pieces of the graph – essentially spines of apply nodes – that represent (parts of) instantiated supercombinator bodies. These pieces are subsequently reduced by systematically overwriting application nodes with values or with graphs of canonical forms.[4]

The valid (or active) context held in the stack includes an activation record (or an argument frame) for the parameters of the supercombinator that is being evaluated, and on top of it some dynamically expanding and collapsing workspace for temporaries. The actual focus of code execution is defined by the topmost instruction in C in conjunction with a pointer to the graph node that it operates on.

Figure 9.1 shows how the G-machine prepares the application

$$(f\ e_1\ e_2\ e_3\ e_4)$$

of a supercombinator f (whose arity is assumed to be three) to four operand expressions $e_1,\ e_2,\ e_3,\ e_4$ for code-controlled execution. The equivalent G-graph derives from the preorder linearized constructor expression

$$@\ @\ @\ @\ \nabla\ 3\ \mathit{code}[f]\ e_1\ e_2\ e_3\ e_4 \quad ,$$

in which ∇ denotes a supercombinator graph node for the G-code of arity 3 that evaluates the instantiated body of f.

The machine sets out with the graph pointer that is on top of the stack, pointing to some topmost apply node (Fig. 9.1(a)), and traverses the spine of the graph until it reaches the node ∇ in the head. While moving down the spine, the graph pointers are stacked up in S which, when arriving at the head, thus contains the complete trace of apply nodes encountered along the spine (Fig. 9.1(b)). Upon inspecting the arity of the ∇ node, the machine reverses gear and, as a prelude to actually executing the supercombinator

[4] An expression graph is said to be in canonical form if its topmost node is something other than an applicator.

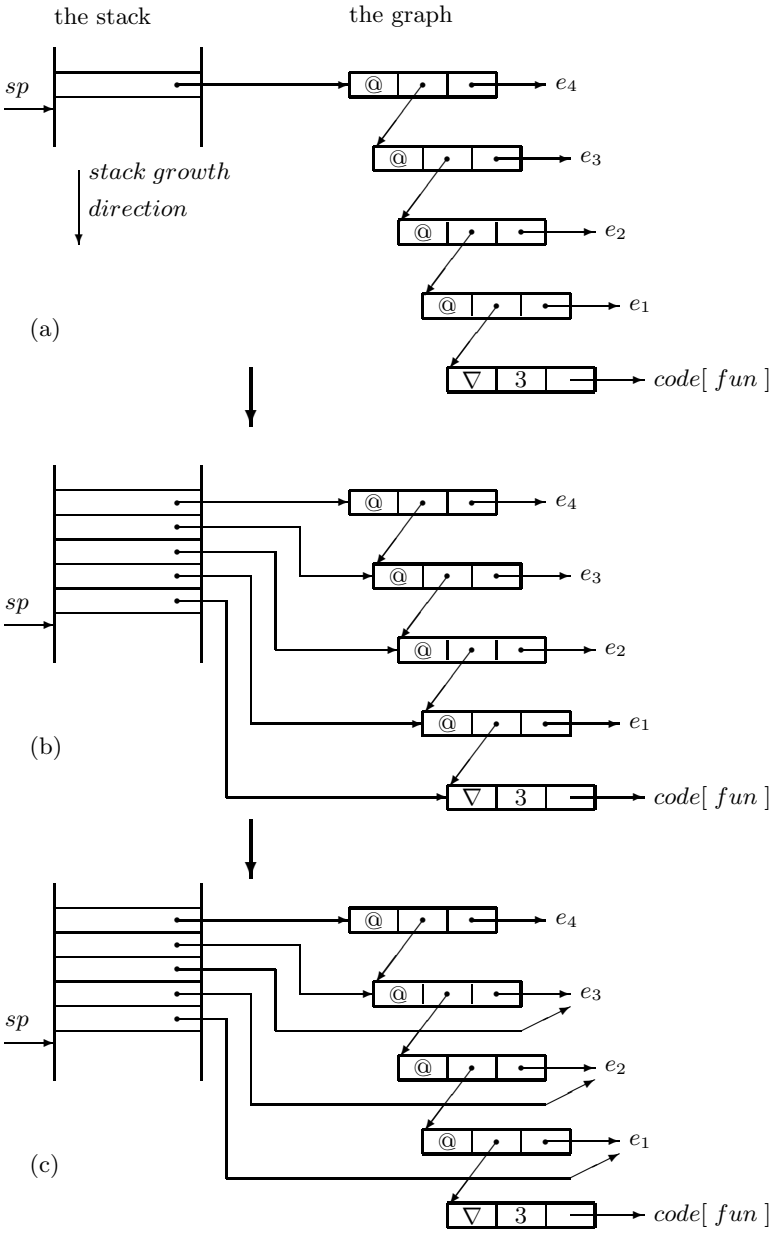


Fig. 9.1. Unwinding a spine on the stack

code, dereferences the three topmost pointers held in S to rearrange the stack so that the tail pointers are placed on top of the pointer to the topmost apply

node of the full supercombinator application (Fig. 9.1(c)). This is the node that eventually must be overwritten with the value (or canonical form) of the application.[5]

The frame of tail pointers thus constructed, together with items that may be temporarily pushed onto S, constitutes the context in which the code $code[f]$ must be executed.

We immediately recognize that running down the pointers of the spine is what the head-order graph reducer of Chap. 7 does as well, with a slight difference though: the *G*-machine completely ignores operand expressions in the tails of the apply nodes, whereas the G_HOR machine wraps them up in suspensions to postpone evaluation in its environment until it becomes absolutely necessary to do so. The *G*-machine delays the evaluation of these expressions too but, as we will see a little later, it instantiates the variables as part of constructing the respective graphs, and it does so right away in the current context.

Other differences relate to the fact that the G_HOR machine (and its descendant, the *B*-machine of Chap. 8) supports a fairly complex environment for full β-reductions compared with simple, coherent contexts for supercombinator reductions and, as a by-product of this simplification, to the skillful merging of the environment stack E, the workspace stack W and the trace stack M of the G_HOR machine and of the *B*-machine in a single stack S of the *G*-machine.

The *G*-machine employs a single control instruction UNWIND to traverse the spine of apply nodes from top to bottom, to dereference from the bottom up as many node pointers stacked up in S as the arity of the function-turned-supercombinator demands, and to rearrange the stack. Thus, UNWIND is in fact a fairly sophisticated graph interpreter of about the same complexity as the part of G_HOR and of the *B*-machine that prepares *apps–lambs* corners for β-reductions.

Stack entries are in the *G*-machine accessed with fixed offsets relative to the current stack top, and only within the topmost context. These offsets are statically figured out by the *G*-compiler based on the binding indices $\#i$ as they occur in the index representations of the supercombinators, and on the number of temporaries that are pushed on top. Since all stack entries are pointers of unit length, the compiler uses

- a parameter d that measures the size of the current context in multiples of these units;
- a mapping ρ that assigns indices $j \in \{0, \ldots, d-1\}$ in monotonically ascending order to all entries of the current context, starting with the index $j = 0$ at the topmost entry, and with the pointer to the topmost application node (the one that needs to be updated by the value of the

[5] The stack pointer *sp* again points to the first empty position on top of the topmost valid entry.

particular supercombinator application) a distance of $d-1$ entries away from the top.

As for supercombinator applications of the general form

$$(\underbrace{\Lambda \ldots \Lambda}_{r}.e_0\ e_1\ e_2\ \ldots\ e_r)\ ,$$

we remember from Sect. 4.3 that in the body term e_0 we may only have occurrences of binding indices $\#i$ with $i \in \{0, \ldots, r-1\}$, and that the indices $\#i = \#0$ and $\#i = \#r - 1$ are bound to the innermost and outermost Λs, respectively, of the *lambs* sequence. Reducing this application thus requires that indices $\#i$ be substituted by operand terms e_{r-i}, i.e., the innermost operand e_1 replaces index occurrences $\#(r-1)$ and the outermost operand e_r replaces index occurrences $\#0$. Looking at the stack configurations of Fig. 9.1, we recognize that the pointer to e_1 is at the stack top and the pointer to e_r (with $r = 3$ in this particular example) is $r - 1 = 2$ entries deeper down in the stack.

The size of the context frame being d, this is also the distance of the deepest entry (the pointer to the topmost apply node) relative to the current stack pointer *sp* (see Fig. 9.1). We can therefore define the function ρ as a mapping of binding indices $\#i$ to offsets j relative to *sp* as

$$j = \rho(\ \#i\ ) = d - 1 - i\ .$$

This mapping may be validated by the following consideration: assuming that the context contains no temporaries, then the size of the context frame is $d = r + 1$, i.e., the number r of parameters plus the additional node pointer at the bottom. Thus, the indices $\#i = \#0$ and $\#i = \#(r-1)$ translate, as expected, into offsets $j = r$ and $j = 1$, respectively. Each temporary that is added increases the size d of the context, and with it the offsets j relative to the stack top, by one.

9.3 Compiling Supercombinators to *G*-Machine Code

We begin the specification of the *G*-compiler without knowing any *G*-machine instructions other than UNWIND, of which we know so far only how it basically works. As we go along, we will introduce other instructions on an informal basis as well, and postpone formal definitions until later. We also ignore typing issues and simply assume that all source programs submitted to the *G*-compiler are well typed.

The *G*-compiler uses three major compilation schemes $\mathcal{F}$, $\mathcal{R}$ and $\mathcal{C}$. The top-level scheme $\mathcal{F}$ applies to supercombinator definitions and transforms them thus:

$$\mathcal{F}[\ f = \underbrace{\Lambda \ldots \Lambda}_{r}.f\_e\ ] \Longrightarrow \text{GLABEL}\ f\ r;\ \mathcal{R}[\ f\_e,\ (\ \rho,\ (r+1)\ )\ ];\ .$$

It generates a pseudoinstruction GLABEL $f\ r$ that assigns to the code produced by $\mathcal{R}$ an entry label (or pointer) p_f under which it may be located in the global code section, together with the number r of argument pointers that the code needs to find in the stack.

$\mathcal{R}[\ f\_e,\ (\ \rho,\ d\ )\ ]$ defines a compilation scheme for supercombinator bodies f_e, with the function ρ and the size d of the context frame as parameters. It generates the code

$$\mathcal{R}[\ f\_e,\ (\ \rho,\ d\ )\ ] \Longrightarrow \mathcal{C}[\ f\_e,\ (\ \rho,\ d\ )\ )\ ];\ \text{UPDATE}\ (d+1);\ \text{POP}\ d;\ \ \text{UNWIND}\ \ ,$$

that breaks down as follows:

$\mathcal{C}[\ f\_e,\ (\ \rho,\ d\ )\ ]$ generates the code for the expression f_e itself. It basically constructs a graph and, upon completing it, leaves a pointer to it on the stack, thus increasing the size of the actual context frame by one.

UPDATE $(d+1)$ overwrites with the pointer pushed by $\mathcal{C}[\ f\_e,\ (\ \rho,\ d\ )\ ]$ the pointer to the topmost apply node of the application of f, which is deeper down in the stack by an offset $d+1$, and subsequently pops this pointer.

POP d pops the topmost d entries off the stack (which is the complete set of arguments of f), leaving only the pointer to the new graph in S.

UNWIND unwinds along its spine the new graph referenced by the topmost pointer on the stack to prepare it for execution of the code found in its head.

The compilation scheme $\mathcal{C}$ applies only to expressions. The code that it produces constructs instantiated graphs using the mapping ρ and the frame size d to compute the offsets of the frame entries by which binding indices $\#i$ must be substituted.

Atomic expressions such as binding indices, function identifiers and constant values are translated by $\mathcal{C}$ into specific PUSH instructions that push onto the stack whatever items follow as parameters, and applications are translated into (sequences of) MKAP instructions that construct apply nodes, with the pointers to the subgraphs taken off the top of the stack (first the pointer to the left subgraph and then the pointer to the right subgraph).

$$\mathcal{C}[e,\ (\rho,d)] \Longrightarrow \begin{cases} \text{PUSHVAL}\ const & \text{if } e =_s\ const \\ \text{PUSHFUN}\ p\_f & \text{if } e =_s\ f \\ \text{PUSHIND}\ \rho(\ \#i\ ) & \text{if } e =_s \#i \\ \mathcal{C}[\ e_1,\ (\rho,\ d)\ ];\mathcal{C}[\ e_0,\ (\rho,\ (d+1)];\ \text{MKAP}; & \text{if } e =_s (e_0\ e_1)\ . \end{cases}$$

Applications $(e_0\ e_1\ e_2\ \ldots\ e_r)$ are by $\mathcal{C}$ treated as r-fold nestings of binary applications $(\ldots((e_0\ e_1)\ e_2)\ldots e_r)$ that compile to

$$\mathcal{C}[\ e_r,\ (\ \rho,\ d\ )\ ];\ldots\ ;\mathcal{C}[\ e_1,\ (\ \rho,\ (d+r-1)\ )\ )\ ];$$

$$\mathcal{C}[\, e_0,\ (\, \rho,\ (d+r)\ )\ ];\ \text{MKAP};\ \text{MKAP};\ \ldots\ ;\ \text{MKAP}\ .$$

This code constructs a spine of apply nodes to the left, with the graphs of the operand expressions branching off to the right, as shown in Fig. 9.1.

The compilation schemes $\mathcal{R}$ and $\mathcal{C}$ may be used to compile the goal expression of a set of supercombinator definitions as well. This expression, by definition, is nothing but the special case of a supercombinator of arity zero since it must not contain any free variables.

The context in which the graph of the goal expression must be reduced contains initially just the pointer to its code, which is what eventually needs to be updated by the resulting graph.

The complete code of a legitimate program expression thus typically includes

- some piece of code that initializes the runtime environment, i.e., the pointers to the stack, the graph and the dump, all of which are initially empty, and calls, through a unique entry label *p_s*,
- the G-code for the goal expression that, in turn, calls
- the G-codes for the various supercombinators, which have assigned to them entry labels by which they may be referenced from within the goal expression and may also cross-reference each other;
- a runtime library for the G-codes that evaluate primitive function applications, i.e., perform δ-reductions.

The top-level initialization and control code of a program is as follows:

$$\text{BEGIN};\ \text{PUSHGLABEL}\ \mathit{ll\_s};\ \text{EVAL};\ \text{PRINT};\ \text{END};\ ,$$

where the instruction

BEGIN initializes the runtime environment;
PUSHGLABEL *ll_s* pushes the pointer (label) to the code of the goal expression onto the stack;
EVAL forces the evaluation of the code referenced by the pointer found on top of the stack and returns the graph of a canonical form;
PRINT forces reduction to normal form of the graph referenced by the pointer on top of the stack and prints it on the output medium;
END terminates code execution.

The instructions EVAL and PRINT play a key role in executing G-programs. They may occur recursively in all codes to control the reduction of selected parts of the graph in compliance with the lazy evaluation regime. In the next section, we will see more of them in the codes that perform various δ-reductions.

9.4 *G*-Code for Primitive Functions

Primitive functions fit perfectly into the picture of the *G*-machine since they are nothing but special (super)combinators: they are trivially closed in the sense that their evaluation does not depend on anything held in an environment, and they need to be applied to as many arguments (of compatible types) as their arities demand to return meaningful results. Full applications of primitive functions may therefore be reduced by ordinary *G*-code held in a runtime library that is linked to each executable program. Occurrences in high-level programs of primitive function symbols such as $+$, $-$, $*$, ..., `gt`, ..., etc. are by the *G*-compiler simply converted into entry labels ll_pf of this library.

Calling the code of a particular primitive function follows the same mechanism as for user-defined functions that have been converted into supercombinators: before the code is entered through the label ll_pf, the application of the function is unwound on the stack by pushing, in the order from innermost to outermost, the argument pointers on top of the topmost apply node.

As the first example, we consider the graph of the application

$$@\ @\ ll\_\texttt{add}\ e_1\ e_2$$

that is supposed to add the values of the expressions e_1 and e_2. Unwinding this application results in a stack configuration in which we have the pointer to e_1 on top of the pointer to e_2 on top of the pointer to the outermost apply node.

Since under a lazy regime the expressions e_1 and e_2 cannot generally be expected to be in normal form but addition is strict in its arguments, requiring numbers, the evaluation of these expressions must be enforced first. Thus, the code to which the label $ll\_\texttt{add}$ refers is as follows:

PUSH 2; EVAL; PUSH 2; EVAL; ADD; UPDATE 3; POP 2; RETURN; ,

where

PUSH i pushes onto the stack the pointer to the expression that can be found by an offset of i positions deeper down in the stack (relative to the stack pointer sp);

EVAL again forces the evaluation of the expression referenced by the topmost pointer in S and returns a graph in canonical form;

ADD adds the two topmost values of the stack, pops them, and pushes the result instead;

UPDATE 3 overwrites with this result value the application node that is by an offset of 3 positions away from the stack top, and subsequently pops this result value;

POP 2 pops both arguments of the application to bring to the stack top the pointer to what originally was the outermost apply node and now points to the result;

RETURN returns control to the calling code.

Another important piece of library code evaluates conditional expressions given by:

$$@\ @\ @\ ll\_\texttt{if}\ e_0\ e_1\ e_2\ \ .$$

Conditionals are strict in their first arguments – the predicate expressions e_0 – and, depending on predicate values, are expected to evaluate either e_1 or e_2 to canonical form. With the pointer to e_0 on top of the pointer to e_1 on top of the pointer to e_2 in S, and with the pointer to the outermost apply node underneath, i.e., an offset of 4 away from the stack top, this may be accomplished by the following piece of code referenced by the label $ll\_\texttt{if}$:

PUSH 1; EVAL; JPFALSE ll_f; PUSH 2; JUMP ll_t; LABEL ll_f;
PUSH 3; LABEL ll_t; EVAL; UPDATE 4; POP 3; UNWIND; ,

where

PUSH 1 pushes the pointer to the predicate expression;
EVAL that follows next evaluates the predicate, leaving a truth value in S; the second occurrence of EVAL evaluates the expression that at this point is referenced from the stack top, which is either e_1 or e_2;
JPFALSE ll_f jumps conditionally to the label ll_f upon finding in S the predicate value `false`, which is popped;
PUSH 2 pushes the pointer to the consequent expression;
JUMP ll_t jumps unconditionally to label ll_t;
LABEL ll is a pseudoinstruction that inserts a label ll in its place;
PUSH 3 pushes the pointer to the alternative expression e_1;
UPDATE 4 overwrites the application node now 4 positions down from the stack top with the canonical form of the conditional;
POP 3 pops the three arguments;
UNWIND continues with further reductions of the graph created by the code of either e_1 or e_2.

This code is structured fairly closely to what compilers of conventional (imperative) languages would produce. The codes for predicate, consequent and alternative expressions, or, more precisely, the EVALs that in fact call them, are simply arranged in sequential order, and depending on the predicate value, code execution jumps around either the (call for the) alternative code or the consequent code.

9.5 The Controlling Instructions ∗

We can now be more specific about the instruction EVAL and about why in some cases we use the instruction RETURN and in others the instruction UNWIND at the end of a piece of code that implements a function application.

The instruction EVAL is the sole enactor of code-controlled reductions in the G-machine, forcing execution of the code referenced from the top of the

stack. It occurs in *G*-code wherever the lazy regime demands evaluation to canonical form of some tail expression along the spine that is unwound on the stack. This is the case for all arguments of strict (primitive) functions occuring in the head position of a spine, and trivially so for the graph generated for the goal expression, which for this purpose may be considered the tail of an application outside the program text itself. The graphs returned by EVAL are generally in canonical form, i.e., the top-level node is something other than an applicator.

Reducing subgraphs just to canonical forms ensures that no more work is done than is absolutely necessary to compute (weak) normal forms. It involves unwinding spines, executing the codes referenced in their heads, which may lead recursively to further unwindings along the spines, and eventually overwriting the root nodes of the (sub)graphs.

Thus, EVAL may be considered a standardized subroutine call that expects to find a single reference parameter – the pointer to some piece of code – on top of the stack. This subroutine begins with the instruction UNWIND to set up in S an argument frame for the code entered at the head of the spine. After completion of the code, control either returns to the calling code by means of the instruction RETURN, or takes a shortcut to continue with further reductions along the spine, using UNWIND as the last instruction.

The choice of the terminating instruction depends on what the code is computing. If it returns as the canonical form some value in a conventional sense, for instance a number or a list, then RETURN is used since nothing else can be done along the spine. However, if the machine executes code that includes recursively supercombinator calls, it must continue to unwind the spine within the current call of EVAL. Of course, UNWIND must also include a RETURN to deal with codes that produce canonical forms other than functions.

EVAL trivially returns its parameter unchanged if this is a constant value, a binary list constructor node $<$ or a function, all of which, by definition, already have canonical form. Similarly, UNWIND does not touch a constant value or a $<$ node since both are already trivially unwound.

A formal definition of the instructions EVAL, UNWIND and RETURN by means of state transition rules is given in Fig. 9.2. These rules use essentially the same notation as in the preceding chapters, and

- val for a constant value,
- $< p_1\ p_2$ for a binary list,
- $@\ p_h\ p_t$ for an apply node,
- $\nabla\ k\ code[\ f\ ]$ for a supercombinator node,

where the p_x are pointers to subgraphs. We use also the symbol G for the heap that holds the graph, and $G[\ p \rightsquigarrow e_1 \mid e_2 \mid \ \ldots\ ]$ to denote alternative (pieces of) graphs to which p may be pointing.

The definition of EVAL requires three rules, of which rule (1), after saving the current machine state in the dump, starts unwinding on the stack a spine of apply nodes, rule (2) starts executing supercombinator code of zero arity

$$(1)\ (\ p : S,\ G[\ p \rightsquigarrow @\ p_h\ p_t\ ],\ \textsc{eval}: C,\ D\ ) \rightarrow (\ p : nil,\ G[],\ \textsc{unwind} : nil,\ (\ S,\ C,\ D\ )\ )\ .$$

$$(2)\ (\ p : S,\ G[\ p \rightsquigarrow \nabla\ 0\ code[\ f\ ]\ ],\ \textsc{eval} : C,\ D\ ) \rightarrow (\ p : nil,\ G[],\ code[\ f\ ] : nil,\ (\ S,\ C,\ D\ )\ )\ .$$

$$(3)\ (\ p : S,\ G[\ p \rightsquigarrow val \mid \nabla\ n\ code[\ f\ ] \mid\ <\ p_1\ p_2\ ],\ \textsc{eval} : C,\ D\ ) \rightarrow (\ p : S,\ G[],\ C,\ D\ )\ .$$

$$(4)\ (\ p : S,\ G[\ p \rightsquigarrow @\ p_h\ p_t\ ],\ \textsc{unwind} : nil,\ D\ ) \rightarrow (\ p_h : p : S,\ G[],\ \textsc{unwind} : nil,\ D\ )\ .$$

$$(5)\ (\ p_0 : p_1 : \ldots p_r : S,\ G[\ p_0 \rightsquigarrow \nabla\ r\ code[\ f\ ] \mid \forall\ i \in \{1, \ldots, r\ }\ p_i \rightsquigarrow @\ p_{h(i-1)}\ p_{t(i)}\ ],\ \textsc{unwind} : nil,\ D\ ) \mid (\ s \geq r\ ) \rightarrow (\ p_{t(1)} : p_{t(2)} : \ldots : p_{t(r)} : S,\ G[],\ C,\ D\ )\ .$$

$$(6)\ (\ p_0 : p_1 : \ldots p_s : nil,\ G[\ p_0 \rightsquigarrow \nabla\ r\ code[\ f\ ]\ ],\ \textsc{unwind} : nil,\ (\ S,\ C,\ D\ )\ ) \mid (\ s < r\ ) \rightarrow (\ p_s : S,\ G[],\ C,\ D\ )\ .$$

$$(7)\ (\ p : nil,\ G[\ p \rightsquigarrow val \mid\ <\ p_1\ p_2\ ],\ \textsc{unwind} : nil,\ (\ S,\ C,\ D\ )\ ) \rightarrow (\ p : S,\ G[],\ C,\ D\ )\ .$$

$$(8)\ (\ p_0 : p_1 : \ldots p_s : nil,\ G,\ \textsc{return} : nil,\ (\ S,\ C,\ D\ )\ ) \rightarrow (\ p_s : S,\ G,\ C,\ D\ )\ .$$

Fig. 9.2. The state transition rules for the G-machine control instructions

(which primarily applies to the goal expression of a program), and rule (3) applies to the trivial cases where the topmost graph node is either a constant value, a list constructor or a supercombinator node of nonzero arity that, without doing anything, let the machine continue with the next instruction in sequence.

The instruction UNWIND breaks down into four rules, of which two handle the 'regular' cases, and the other two continue with the machine state saved in the dump if no reductions can be performed on the spine. Rule (4) pushes the pointer to an apply node and continues unwinding the spine along its head pointer p_h. When encountering in the head of a spine a supercombinator node of arity r, rule (5) dereferences the topmost r pointers, provided there are as many, and replaces them with the tail pointers of the respective apply nodes. Rule (6) terminates UNWIND in the case of a partial application, clearing the stack down to the topmost apply node of the unwound spine, and rule (7) does nothing if the topmost stack entry is either a constant value or a list

constructor. Both of the latter two rules unsave the machine state that is in the dump.

Finally, rule (8) defines the instruction RETURN. It effects continuation with the machine state saved in the dump in all cases that are not implicitly taken care of by UNWIND, but restores the lowermost pointer on the old stack as the top pointer on the new stack in order to pass the result of some function application on to the calling code. This instruction is mainly used to return from primitive function code such as that for addition.

Implementing these instructions is simpler than the state transition rules may suggest, particularly with regard to the dump. What is to be saved and unsaved in it are conceptually tuples $(S,\ C)$. However, looking at the rules for EVAL and RETURN, we note that the graph pointer p in the topmost position of the old stack becomes the lowermost entry of the new stack, and vice versa. Thus, rather than saving the stack in another structure, the dump, it may simply be extended, and the contexts of the calling and the called codes may be overlapped with regard to this entry – a very convenient way of passing the graph pointer back and forth between the two. This leaves the actual state of execution of the calling code to be saved in the dump. Since in an implementation on a real machine this state is represented by an instruction counter that points to the instruction immediately following EVAL, the dump can in fact be reduced to a stack for return addresses.

The last control instruction to be defined is PRINT. It is used in the initialization code to force the reduction to weak normal form of the canonical form of the goal expression returned by the instruction EVAL, and to print this weak normal form on the output medium. In the original *G*-machine PRINT produces legitimate output only for (sequences of) atomic values, but not for normal forms that include functions. Since the *G*-machine compiles functions to code that has generally undergone several optimizations, and from which all variables are gone, they cannot be decompiled anymore into intelligible high-level output that bears any resemblance to the original definitions. Also, partial function applications are left as they are and not returned as output.

To include the output in the definition of PRINT, we need to extend the state of the *G*-machine by another component O to get

$$(\ O,\ p : S,\ G[\ p \rightsquigarrow\ val\ ],\ \text{PRINT} : C,\ D\ )\ \rightarrow (\ O : val,\ S,\ G[],\ C,\ D\ )$$

for atomic values, and

$$(\ O,\ p : S,\ G[\ p \rightsquigarrow <\ p_1\ p_2\ ],\ \text{PRINT} : C,\ D\ )\ \rightarrow \\ (\ O,\ p_1 : p_2 : S,\ G[],\ \text{EVAL} : \text{PRINT} : \text{EVAL} : \text{PRINT} : C,\ D\ )$$

for lists, in which case the subgraphs need to be first evaluated and then printed.

In all other cases, the output either remains unchanged or produces some error message.

It may also be noted that PRINT, as defined here, outputs the elements of a list in flat form, i.e., without preserving its structure. However, this is a minor problem that may be easily fixed.

9.6 Some *G*-Code Optimizations

Abstract machines, as intermediate levels of compilation, provide a platform for a variety of code optimizations that are invariant against the choice of a particular target architecture.

Primary targets for G-code optimizations are sequences of costly MKAP instructions that construct superfluous application nodes which subsequently have to be reduced by equally superfluous UNWINDs, UPDATEs and even EVALs. They come about when compiling the bodies of supercombinator, as in

$$f = \lambda u_1 u_2 \ldots u_r.(g\ e_1\ e_2 \ldots e_n)\ ,$$

using the compilation scheme $\mathcal{R}$ without regard for what the function g really looks like:

$$\begin{aligned}&\mathcal{R}[\ (g\ e_1\ e_2 \ldots e_n),\ (\ \rho,\ d\ )\ ] \Longrightarrow\\ &\qquad \mathcal{C}[\ e_n,\ (\ \rho,\ d\ )\ ]; \ldots ; \mathcal{C}[\ e_1,\ (\ \rho,\ (d+n-1)\ )\ ];\ \mathcal{C}[\ g,\ (\ \rho,\ (d+n)\ )\ ];\\ &\qquad\qquad \text{MKAP; } \ldots \text{ ; MKAP; UPDATE } (d{+}1); \text{ POP } d; \text{ UNWIND;}\end{aligned}$$

The code thus obtained routinely goes through the motions of

- constructing an instantiated graph in the context created by the application of f;
- updating the root node of the original graph with it and then popping the arguments of f;
- then unwinding the spine of the new graph, and thereby often pushing again what was already or still is on the stack;
- and finally, executing the code for the function g, provided the spine holds enough arguments.

However, there are many applications whose evaluation, if more cleverly organized, could do without constructing and unwinding new spines.

Typical examples are applications of the primitive functions + and `if` to just the right number of arguments. These applications could be compiled by $\mathcal{R}$ without using MKAP instructions as:

$$\begin{aligned}&\mathcal{R}[\ (\ +\ e_1\ e_2\ ),\ (\ \rho,\ d\ )\ ] \Longrightarrow\\ &\qquad \mathcal{C}[\ e_2,\ (\ \rho,\ d\ )\ ]; \text{ EVAL; } \mathcal{C}[\ e_1,\ (\ \rho,\ (d+1)\ )\ ];\\ &\qquad\qquad \text{EVAL; ADD; UPDATE } (d{+}1); \text{ POP } d; \text{ RETURN;}\end{aligned}$$

$$\mathcal{R}[\ (\texttt{if}\ e_0\ e_1\ e_2),\ (\ \rho,\ d\ )\ ] \Longrightarrow$$

$$\mathcal{C}[\, e_0,\ (\, \rho,\ d\,)\,];\ \textsc{eval};\ \textsc{jumpfalse}\ \ ll\_f;$$
$$\mathcal{R}[\, e_1,\ (\, \rho,\ d\,)\,];\ \textsc{label}\ ll\_f;\ \ \mathcal{R}[\, e_2,\ (\, \rho,\ d\,)\,];\ \ .$$

Also, the UNWINDs are replaced by RETURNs in the code for +, and terminating both the consequent and alternative codes of the primitive `if` with RETURNs, whenever applicable, is ensured by recursive application of the $\mathcal{R}$-scheme.

A very effective optimization may be applied if the function g called in the body of the above supercombinator f is another (or the same) supercombinator, i.e., we have a tail (recursive) function call. Since there is nothing left to do in the body of f once control returns from g, we may at least expect to have the context created by the application of f released before g is entered and thus save space in the stack. But there is a little more to this.

When reducing a full application of f to, say, arguments $a_1, \ \ldots\ , a_r$, the code for f generated by a nonoptimizing scheme $\mathcal{R}$ first creates a context for the evaluation of f's body by pushing the argument pointers, and then continues to execute the code $\mathcal{C}[\, e_n\,];\ \ldots\ ;\ \mathcal{C}[\, e_1\,];$. This extends the context of f by pointers to the graphs created for $e_1, \ldots,\ e_n$, as shown in Fig. 9.3(a). Up to this point, the optimized code has to do the same as the nonoptimized code.

However, the nonoptimized code would now execute a sequence of n MKAP instructions to construct a complete graph for the application $(g\ e_1\ \ldots\ e_n)$. UPDATE $d+1$ and POP d would overwrite the root node of the application of f and then clear the argument pointers off the stack, as in the configuration of Fig. 9.3(b). Finally, UNWIND would unwind the spine of this graph again to bring about the configuration shown in Fig. 9.3(c).

We recognize that, with regard to the application of g to its arguments $e_1\ \ldots\ e_n$, we had the same configuration on the stack already as part of the context of f. Also, updating the root node of the application of f can obviously be dispensed with since the resulting graph is in fact constructed by the code for g that updates the very same node. So, we may take a shortcut to get from the stack configuration of Fig. 9.3(a) to that of Fig. 9.3(c) by squeezing the r argument pointers of f out from underneath the n argument pointers for g (plus the pointer to g itself), and then branch directly to the code for g.

Thus, the optimized code to be generated by $\mathcal{R}$ must look like this:

$$\mathcal{R}[\, (g\ e_1\ \ldots\ e_n),\ (\, \rho,\ d\,)\,] \Longrightarrow$$
$$\mathcal{C}[\, e_n,\ (\, \rho,\ d\,)\,];\ldots;\ \mathcal{C}[\, e_1,\ (\, \rho,\ (d+n-1)\,)\,];$$
$$\textsc{pushlabel}\ ll\_g;\ \textsc{squeeze}\ (n+1)\ r;\ \textsc{jfun};\ \ ,$$

where SQUEEZE does the squeezing and JFUN takes the topmost stack entry as the code label ll_g. This code passes the arguments of g directly, i.e., without going through the redundant moves of constructing a graph from the

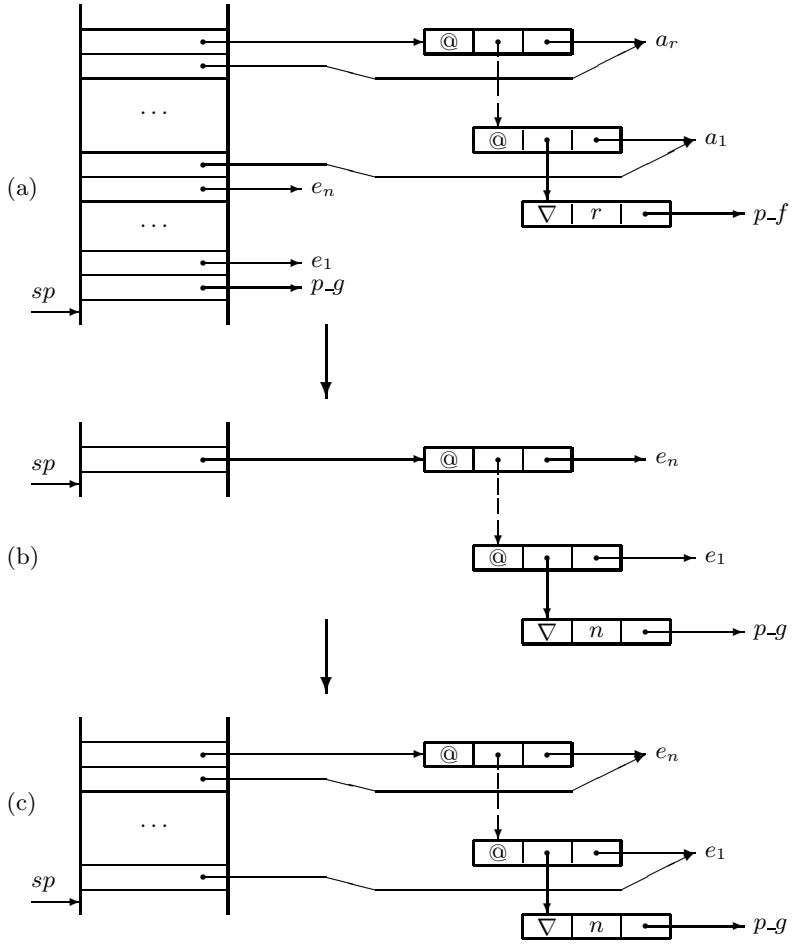


Fig. 9.3. Handling tail-end recursions in the *G*-machine

stack entries $e_1 \ldots e_n$ just to have them immediately restored by unwinding this graph again.[6]

9.7 Summary

The idea of the *G*-machine is to perform head-order graph reductions by compiled code whose raw runtime performance is in the same league as that of code obtained by compilation of conventional (procedural) programs. To

[6] More *G*-code optimizations are described in the original papers by Johnsson or in the textbook by Peyton Jones.

achieve this end, some restrictions have to be imposed on the language accepted by the compiler. They constitute a fairly radical departure from the full-fledged λ-calculus as supported by the reduction machines of the preceding chapters. The machine is just weakly normalizing, permitting merely the computation of ground terms (or basic values).

Compiling user-defined functions to efficient code means that they become static objects that need to be supplied with full sets of arguments in order to execute correctly, with the consequence that they lose their status as first-class objects. Functions may still be applied to functions, but there is no way of computing new functions, say, as results of partial applications. What can be done though is to pass partial applications along as they are, hoping that they may pick up the missing arguments in the course of subsequent reductions, but they cannot be turned into intelligible output. Similarly, variables may only be used as identifiers but there is no way of supporting them as first-class objects (or as their own values) either. Last but not least, compilation to free-running code also demands that programs be well typed so that no type inconsistencies can occur at runtime.

The approach taken with the *G*-machine, therefore, is to convert, prior to compilation, nested sets of function definitions into flat sets of supercombinators (or closed λ-abstractions), thus systematically eliminating variables that are locally free but bound in larger contexts. The goal expressions of programs must be supercombinators of arity zero, i.e., they must not contain any free variables either. Conversion to supercombinators in fact kills two birds with one stone: all substitutions of formal by actual function parameters can be done naively and in one conceptual step since there can be no name clashes, and compilation yields fairly efficient codes whose active runtime environments (or contexts) are completely contained in single contiguous frames whose entries can be accessed with fixed offsets.

Supercombinator reduction is inherently weakly normalizing since it rules out substitutions and reductions under abstractors.

The *G*-machine works with three runtime structures: a stack, a graph and a dump. Starting with empty structures, the machine repeatedly goes through the basic cycle of having the code construct a graph of application nodes, unwinding the spine of this graph on the stack, attempting to apply the function code referenced in the head of the spine to the arguments found on the stack and, if successful, to overwrite the topmost application node of the unwound graph with the result. Depending on what the function actually is, this result may either be an atomic value or another graph constructed by the code.

In comparison with the *B*-machine of Chap. 8 there does not seem to be much difference regarding this basic cycle. It does not really matter how the code unwinds a spine and subsequently constructs a normalized expression, the effects are basically the same. The *B*-machine does the unwinding by executing forward code consisting of LAM and AP instructions, which generates the equivalent of an unwound spine as backward code composed of AP in-

structions whose parameters are suspensions. The *G*-machine does the same by means of a single instruction UNWIND, which leaves a trace of argument pointers on the stack.

Applying abstractions is in the *B*-machine effected by LAM instructions. They consume AP instructions from the backward code to create environments in which the forward-executing abstraction code generates new backward code. This is equivalent to function-turned-supercombinator calls of the *G*-machine that consume arguments from the stack to construct new spines, which are subsequently unwound on the stack again.

References

The *G*-machine concept was first published by Johnsson in [Joh84], with a separate paper addressing λ-lifting [Joh85]. It is also the subject of his PhD thesis [Joh87]. The machine is also described in various textbooks, for example [PeyJ87, Kog91] and, more recently, [HaMi99]. Supercombinators were first introduced by Hughes in [Hu82a].

Basically the same machinery is used by Turner's SKI-combinator reduction approach for the implementation of MIRANDA [Tur79, Tur85]. A descendant of the *G*-machine – the spineless tagless *G*-machine – is due to Peyton Jones [PeSa89, PeyJ92]. Another spineless graph reduction machine is the three instruction machine TIM by Fairbairn and Wray [FW87].

A code-executing, weakly normalizing abstract graph rewriting machine *ABC* for the functional language CLEAN is described by Plasmeijer and van Eekelen in [PvE93].

Another weakly normalizing machine for an applied λ-calculus, based on sophisticated graph-rewriting techniques that rigorously exploit sharing, is the Bologna Optimal Higher-order Machine by Asperti, Giovanetti and Naleito [AGN96].

There is also a large body of research on parallel versions of graph reducing machines that is summarized in [HaMi99].

10

The π–RED Machinery

In this chapter, we introduce a compiled graph reduction system for a fully normalizing applied λ-calculus. It comes in a lazy (or operands-when-needed) and a strict (or operands-first) version, both of which are based on code-executing abstract machines very similar to the G-machine. They reduce λ-expressions to weak (head) normal forms and cross the borderline toward full normalization by means of an η-extension mechanism that, by interpretation, turns partially applied or unapplied abstractions into full applications to enable further code execution in the abstraction bodies. This approach has the same effect of moving substitutions across abstractors as the *beta*-rule of the $\lambda\sigma$-machine of Sect. 6.3.

π–RED accepts high-level programs of the AL variety as input and compiles them to abstract machine code that may either be interpreted or, in another step, compiled to conventional target machine code. Code execution produces a graph that may be decompiled and returned to the user in the same AL-like high-level notation in which the original program was written. As a distinguishing feature, π–RED also supports an interactively controlled stepwise execution mode. This mode renders it possible to set breakpoints that stop the machine after some prespecified number of β-reductions and to decompile the intermediate machine states reached at these points into high-level program expressions. These expressions may even be modified before they are resubmitted for more reductions. As far as its appearance to the user is concerned, π–RED thus performs high-level program transformations as described in Chaps. 2 and 3, though something different is going on at the machine level.

10.1 The Basic Program Execution Cycle

Compiled graph reduction for two reasons calls for a supercombinator-based parameter-passing mechanism, as in the G-machine, or for some equivalent mechanism of closing λ-expressions. On the one hand, it derives its efficiency largely from substituting λ-bound variables naively by argument pointers,

which means that these substitutions must be kept free of potential name clashes. On the other hand, compiled function code must be supplied with full sets of arguments to execute correctly. Both demands can be satisfied only by systematically closing all abstractions and by treating partial applications as irreducible, the latter being exactly where weak normalization stops.

However, static conversion to supercombinators, or alternatively, turning abstractions into closures at runtime, as the SE(M)CD machine does, inflicts some degree of redundancy, though much of it can be eliminated by subsequent compiler optimization. Supercombinators repeatedly copy the same instantiations of what were originally (relatively) free variables into function calls. Closures must be formed individually for all abstractions defined in local contexts, even if they share the same subsets of free variables, which means that again the same variable instantiations may have to be copied repeatedly.

π–RED employs a less rigorous concept of closing λ-expressions that has this redundancy largely eliminated by a preprocessor. Free variables are systematically lifted (or abstracted) out of the larger contexts of AL-like `letrec` expressions that define sets of mutually recursive functions.[1]

Let e denote a λ-expression whose set of free variables is, say, $\{w_1, \ldots, w_q\}$, then λ-lifting turns e into

$$(\sim \tilde{\lambda} w_1 \ldots w_q . e \; w_1 \ldots w_q) \quad ,$$

where the tilde superscript (or tag) $\sim$ distinguishes the abstractions and applications introduced by lifting from those of the original expression. They will in the following be referred to as tilde abstractions and tilde applications, respectively. λ-lifting implies that tilde abstractions can occur only in operator positions of tilde applications, and that no other expressions can occur in these positions since there is no other way for tilde applications to come about.

Tilde annotations are not part of the high-level syntax and therefore must occur neither in input expressions nor in partially or fully reduced output expressions. They can be introduced only by the preprocessor, and must be removed by a complementary postprocessor before returning legitimate high-level expressions of the language as output.

To illustrate this kind of λ-lifting, consider the nested `letrec` expression[2]

```
letrec
f = λuv.letrec
          g = λwz.if (gt u w) then(g (− 1 u) z) else (f v (+ 1 w))
          in (g v u)
in (f 1 2) .
```

[1] Anonymous abstractions that may occur anywhere within a program expression must be closed individually.

[2] Throughout this chapter we deviate slightly from AL syntax insofar as we use the symbol λ instead of `lambda`.

This expression is composed of two mutually recursive functions, of which g is local to f, using u and v as relatively free variables. λ-lifting changes this expression syntactically to

$$
\begin{array}{l}
\texttt{letrec} \\
f = \lambda uv.(\sim \tilde{\lambda} uv.\texttt{letrec} \\
\qquad\qquad\qquad\qquad g = \lambda wz.\texttt{if}\ (\texttt{gt}\ u\ w)\ \texttt{then}(g\ (-\ 1u)\ z) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ (f\ v\ (+\ 1\ w)) \\
\qquad\qquad\qquad\quad \texttt{in}\ (g\ v\ u) \\
\qquad\qquad\quad u\ v) \\
\texttt{in}\ (f\ 1\ 2)\ .
\end{array}
$$

This expression obviously includes an opportunity for a simple but effective optimization that can be directly carried out by the preprocessor. Since the inner `letrec` in this particular case is the entire body of the abstraction f, and f is defined at top level, the λ-bound variables of f are the same as those that have been lifted, which just introduces an additional parameter-passing step with nothing being done in between. This step can therefore be eliminated, yielding

$$
\begin{array}{l}
\texttt{letrec} \\
f = \tilde{\lambda} uv.\texttt{letrec} \\
\qquad\qquad\quad g = \lambda wz.\texttt{if}\ (\texttt{gt}\ u\ w)\ \texttt{then}(g\ (-\ 1\ u)\ z)\ \texttt{else}\ (\sim f\ v\ (+\ 1\ w)) \\
\qquad\qquad \texttt{in}\ (g\ v\ u) \\
\texttt{in}\ (\sim f\ 1\ 2)\ .
\end{array}
$$

Since f has thus been turned into a tilde abstraction, applications of f must accordingly be converted into tilde applications as well in order to conform to what has been said before about legitimate syntactical positions of tilde abstractions. However, this optimization can only be done if f occurs in operator positions of full applications, as in this particular case. If f occurs as operator of a partial application or is passed along as an operand, then the nonoptimized variant of f must be used since it cannot generally be decided statically whether and where f will be applied eventually.

Another preprocessing step converts all occurrences of λ- and $\tilde{\lambda}$-bound variables into differently tagged reverse binding indices (or binding levels) to prepare the expression for compilation to abstract machine code. This conversion is defined on λ-abstractions as

$$\lambda u_1 u_2 \ldots u_n.e \ \rightarrow\ \Lambda_{u_1} \Lambda_{u_2} \ldots \Lambda_{u_n}.\#e\ ,$$

where $\Lambda_{u_1} \Lambda_{u_2} \ldots \Lambda_{u_n}$ is a sequence of our well-known nameless abstractors,[3] and $\#e$ emerges from e by replacing, for all $i \in \{1, \ldots,\ n\}$, free occurrences of

[3] The original variable names remain attached to the Λ-abstractors as subscripts to enable the compiler to prepare a graph node from which the postprocessor can

u_i with indices $\#(i-1)$. Thus the variable bound to the outermost λ receives the lowest index, and the variable bound to the innermost λ receives the highest index.

An equivalent conversion applies to $\tilde{\lambda}$-abstractions $\tilde{\lambda}w_1 \ldots \tilde{\lambda}w_q.e$, whereby all $\tilde{\lambda}w_j$ are replaced by nameless abstractors $\tilde{\Lambda}_{w_j}$ and free occurrences of all w_j in e are replaced by indices $\sim (j-1)$.[4]

With these conversions, our expression becomes:

```
letrec
f = Λ̃u Λ̃v.letrec
              g = Λw Λz.if (gt ~0 #0) then (g (- 1 ~0) #1)
                                      else (~ f ~1 (+ 1 #0))
            in (g ~1 ~0)
in (~ f 1 2) .
```

All occurrences of the $\tilde{\lambda}$-bound variables u and v, i.e., those that have been lifted, are now replaced by tilde indices ~ 0 and ~ 1, and all occurrences of the λ-bound variables w and z are now replaced by indices $\#0$ and $\#1$, respectively.

Once these these preprocessing steps have been performed, the compiler takes over to translate expressions into abstract machine code for execution. As code execution is just weakly normalizing, the machine may repeatedly call for η-extensions of partially applied or unapplied abstractions that pop to top level to turn them into full applications and thus enable further weakly normalizing code execution phases, until full normal forms are reached. The graph representations of these normal forms are handed over to the postprocessor for reconversion into legitimate high-level output. These program execution phases are graphically depicted in Fig. 10.1.

Reducing expressions in a stepwise mode and returning intermediate expressions as high-level output renders it possible to modify the expressions before resubmission for another sequence of reduction steps. For instance, one could correct programming errors, add new to existing functions, replace subexpressions by others, build new applications, change variable names or the scope of variable bindings, introduce new variables out of thin air, and so on, which of course would also change the meaning of the original program. Moreover, the focus of control may be shifted within the expression to select any subexpression for execution. Referential transparency is nevertheless guaranteed since variables that may occur (relatively) free in these subexpressions but are bound in larger contexts are correctly treated by full-fledged β-reductions.

---

reconstruct the original abstraction, with all variables as introduced by the user, should it become part of the output expression.

[4] In what follows, we will refer to λ-abstractions and $\tilde{\lambda}$-abstractions also as ordinary and tilde abstractions, respectively.

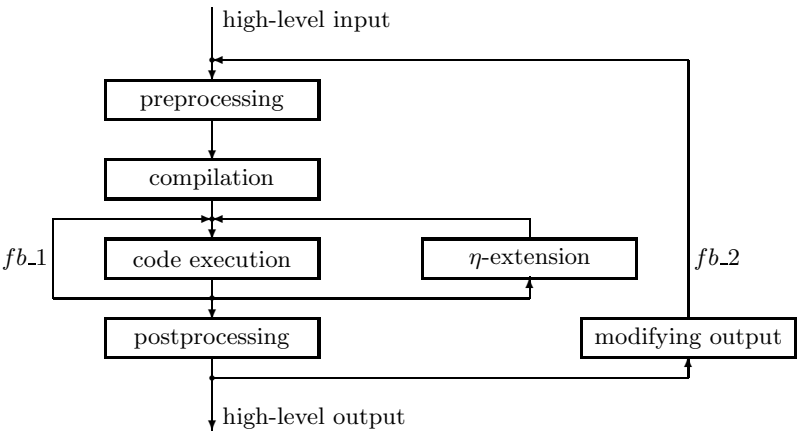


Fig. 10.1. The program execution phases of π–RED

Both of these features are very helpful when, during a program development phase, it comes to validating correctness and correct execution. In Appendix B it will be shown that they play an important role in theorem proving as well.

If either of these options is chosen, the expression at hand must pass through a full execution cycle, including preprocessing and compilation, as depicted by the feedback path fb_2 in Fig. 10.1. If the output is just inspected without modification, then code execution may resume directly with the internal machine state reached at the breakpoint, as indicated by the feedback path fb_1.

Figure 10.2 shows how the effects of three consecutive function calls on our example expression can be visualized as π–RED output when executing them step by step. Each step yields a new expression that results from expanding the function application in what is actually the body of the outermost `letrec` by the instantiated right-hand side of the function definition, and then reducing this expression to the point where the other `letrec` becomes top level.

We note that the example expression has the interesting property of reproducing itself after three function calls with the parameters of the outermost application of f interchanged. Another two such steps restore the original expression, which means that the computation, if left running, never terminates.

Runaway recursions can be taken care of by the same mechanism that controls stepwise execution. We simply need to equip the machine with a counting device for β-reductions or, more appropriately, for entire function calls. Before submitting an expression for evaluation, this counter must be initialized with some integer value greater than zero that defines an upper bound on the number of function calls executed. Each function call decrements this value by one, and code execution stops either after the counter is down

```
letrec
  f = λuv.letrec
            g = λwz.if ( gt u w ) then ( g ( − 1 u ) z )
                                  else ( f v ( + 1 w ) )
          in ( g v u )
in ( f 1 2 )
```

⇓

```
letrec
  g = λwz.if ( gt 1 w ) then ( g ( − 1 1 ) z )
                        else letrec
                               f = ...
                             in ( f 2 ( + 1 w ) )
in ( g 2 1 )
```

⇓

```
letrec
  g = λwz.if ( gt 1 w ) then ( g ( − 1 1 ) z )
                        else letrec
                               f = ...
                             in ( f 2 ( + 1 w ) )
in ( g 0 1 )
```

⇓

```
letrec
  f = λuv.letrec
            g = λwz.if ( gt u w ) then ( g ( − 1 u ) z )
                                  else ( f v ( + 1 w ) )
          in ( g v u )
in ( f 2 1 )
```

Fig. 10.2. Stepwise program execution in π–RED

to zero or after a full normal form has been reached, whichever occurs first. The machine state at this point is in either case reconverted into a high-level expression and returned as output. If the output expression is not yet fully normalized, it may be resubmitted for another sequence of reductions. If the expression is known to terminate, then ensuring that the machine reaches its normal form in one run is just a matter of choosing the initial counter value generously enough.

10.2 The Operating Principles of the Abstract Machines

Both the lazy and the strict code-executing abstract machines use basically the same internal program representation and the same runtime structures, although there are some differences with regard to the latter that concern the representation of the objects they are to accommodate.

Being able to reconstruct complete intermediate program expressions from the machine states left at the breakpoints of stepwise code execution requires that all information about the original program which is getting lost when compiling it must be saved as part of a persistent structure that survives code execution. This information must include the nesting of function definitions in `letrec` expressions, the list of function names, and the lists of bound variable names that the preprocessor has changed into #- or ∼-tagged indices.

Figure 10.3 shows such a structure for a typical `letrec` expression,

$$\tilde{\lambda} w_1 \ldots w_s . \texttt{letrec} \ \ldots \ f_i = \lambda u_1 \ldots u_r . e_i \ldots \texttt{in} \ e_0 \ ,$$

from which the (relatively) free variables $w_1, \ldots, w_q$ have been lifted out.

In this structure, both the entire `letrec` construct and each individual function are represented by descriptors that are distinguished by the tags `letrec` and `func`, respectively. The `letrec` descriptor includes links to the code of the body expression e_0, to the list of free variable names that have been abstracted out, to the list of function names, and to a list of pointers to function descriptors that, in turn, have links to the function codes and to the lists of λ-bound variables. There are also pointers back to the `letrec` under which the functions are defined.

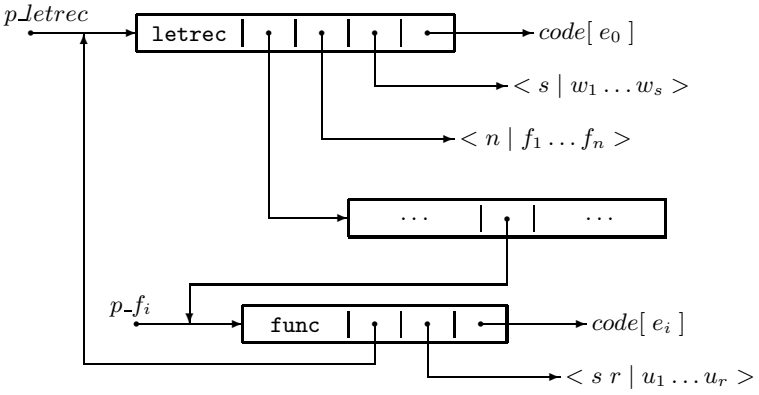


Fig. 10.3. A typical graph structure for a `letrec` expression

The function codes and the code for the `letrec` body contain, as parameters of function calls, only direct pointers p_f_i to function descriptors. These pointers must be dereferenced to branch to the respective codes. The `letrec` descriptors are never referenced during code execution; they are used only by the postprocessor to reconstruct output expressions that include the complete defining contexts of the functions that are still relevant; hence the `letrec` pointers in the function descriptors.[5]

Both abstract machines are defined by tuples $(\ C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ )$ whose components denote from left to right

- the piece of code C actually selected for execution from a graph structure as in Fig. 10.3;
- a system of four runtime stacks $(\ A,\ W,\ T,\ R\ )$;
- a heap H to accommodate graph structures, including the complete code structure;
- a reduction counter value rr that specifies the number of function calls left to be performed.

Distributing the changeable parts of the runtime environment over four stacks simplifies code optimizations, and it also liberates the compiler from calculating offsets relative to changing stack tops – something that may have to be completely redone when target machine code is generated.

The stacks are used as follows:

- stack W serves as a value stack for temporaries and as a buffer space in which the argument frames of function calls build up;
- stack A holds completed argument frames, also called A-frames, for instantiations of Λ-bound reverse binding indices $\#i$, that have been created by full applications of λ-abstractions;
- stack T holds the frames for instantiations of $\tilde{\Lambda}$-bound reverse binding indices $\sim j$, also called T-frames, that result from the reduction of tilde applications;
- stack R accommodates the return continuations of function calls.

In reference to the stacks that hold their instantiations, we will alternatively call the #- and ∼-tagged indices the A- and T-indices, respectively.

Given a nontrivial program that typically features an outermost `letrec` construct for function definitions, the machine starts with the code for its body expression that is held in the structure C, and from there calls the code for applications of top-level functions that, in turn, may recursively call functions defined deeper down in local `letrecs`. The code for a full application of a λ-abstraction defined under a `letrec` builds up the arguments one by one in the workspace stack W and, upon entering the code of the abstraction body, moves them in one conceptual step into a frame in stack A. It also saves the

[5] For reasons of consistency, we will also use this descriptor concept for the changeable parts of the graph and for the environment frames.

remaining code in C as return continuation on stack R, and copies the function code into C instead. The code for a tilde application goes through essentially the same motions, except that it creates a frame in stack T. Occurrences of A-indices or T-indices in the code are used as offsets into the topmost A- or T-frames, respectively, and the entries found under these offsets are subsequently pushed onto the workspace stack W. Completing the code of a Λ-abstraction releases the topmost frame from stack A, and completing the code of a $\tilde{\Lambda}$-abstraction releases the topmost frame from stack T, and in both cases the return continuation is retrieved from R and restored in C.

The complete environment for executing a function call is thus contained in what are currently the topmost A- and T-frames. On the one hand, there is no linking of frames, as for instance in the B-machine, on the other hand, there is no redundancy in passing nonlocal parameters between recursive function calls within the same `letrec` context, as in the G-machine. So we have in fact the simplicity of supercombinator reduction without closing individual `letrec`-defined abstractions.

These basic operating principles are what both the lazy and the strict abstract machine have in common. Beyond that, there are some significant differences regarding the instruction sets and their interpretation, the compilation of preprocessed λ-expressions to machine code, the treatment of operand expressions, and the intricacies of using the runtime stacks. We therefore have to look at the two machines separately, beginning with the more sophisticated lazy version.

10.3 The Lazy Abstract Stack Machine LASM

We begin the description of the LASM with the runtime environment that is distributed over the stack system and the heap, and with the graph structures that are created and operated on as code execution proceeds.

Lazy evaluation applies functions to arguments in unevaluated form and exploits sharing as much as possible to contract redices at most once and only to the extent needed. This means that arguments (or operands) must be wrapped up in suspensions and passed around in this form until a demand for their evaluation arises. Sharing requires that suspensions be represented as graph nodes, just as in the G_HOR and B-machines of Chaps. 7 and 8. The pointers to these nodes may be copied, rearranged or deleted in the course of β-reductions. The environments that need to be included in the suspensions at the time of creation are the topmost A- and T-frames. However, as the suspensions usually survive the contexts, i.e., the function calls in which they come into existence, so must the respective T- and A-frames. This means that they cannot simply be stacked up on top of each other as applications of $\tilde{\lambda}$- or λ-abstractions are nested inside each other (and released in reverse order) but must be held in the heap and referenced from the stacks T and A by pointers. The very same pointers can then also be used to represent these frames in

suspensions. The frames can only be released if they are referenced neither by suspensions nor from the respective stack; likewise, suspensions can be released only if they are not referenced anymore from within stack frames of either kind.

Representing all heap objects by descriptors of the same formats also facilitates the earliest possible release of unused heap space. All it takes to do so is to include reference counters in the descriptors that are incremented whenever pointers to them are replicated, and decremented whenever such pointers are deleted. Once the reference count has come down to zero, both the descriptor slot and the space held by the object itself can be immediately set free (see Sect. 7.5).

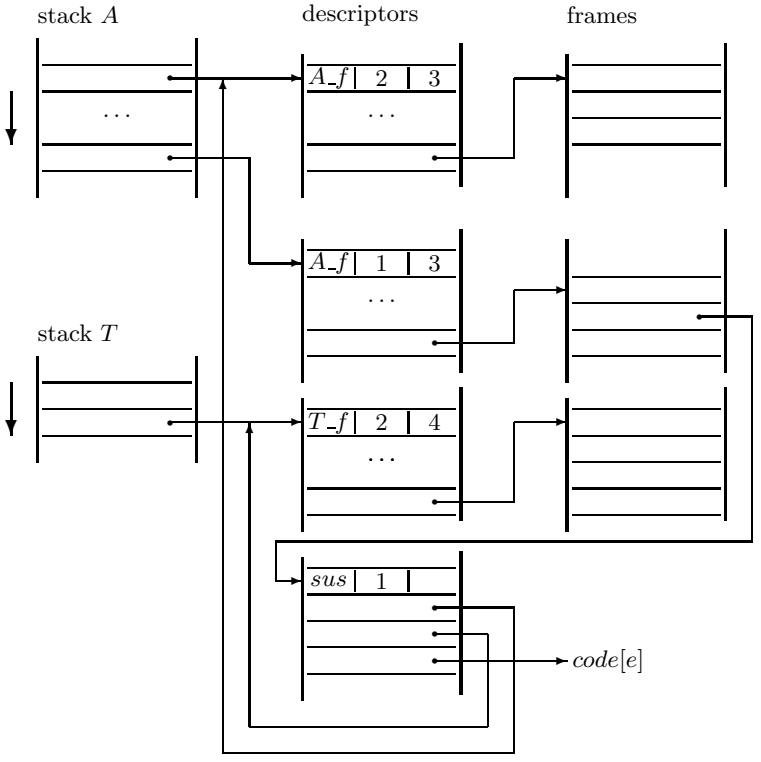


Fig. 10.4. A typical stack | heap configuration of the LASM

Figure 10.4 shows a typical stack | heap configuration as it may develop when executing LASM code. Here we have the stacks on the left, growing downward, whose entries are pointers to frame descriptors, of which three are shown in the middle, two for A-frames and one for a T-frame. Each descriptor is assumed to consist of four words, of which the first one has entries for the

frame type (A_f for an A-frame and T_f for a T-frame), for the reference count (the number of pointers pointing to it) and for the number of argument pointers in the frame itself. The pointers to the frames can be found in the last words of the descriptor formats (the remaining words are not used); the frames themselves are shown on the right. One of the A-frame entries is shown as pointing to the descriptor of a suspension (at the bottom of the descriptor column) with a reference count of one, one pointer each to the A- and T-frames that define the environment, and a pointer to the code that needs to be executed in this environment.

10.3.1 The LASM Instruction Set

Now that we have an idea of what the runtime structures basically look like and how they need to be operated on, we are ready to specify the LASM instructions that are necessary to implement function applications:

PUSH_C *value* takes a basic value as its parameter, turns it into a heap object and pushes a pointer to it onto stack W.

PUSH_P *pp* takes a pointer *pp* as its parameter and pushes it onto stack W.

PUSH_FUN *prf* pushes a primitive function symbol such as $+$, $-$, $*$, etc. directly onto stack W.

MKSUSP p_e creates a suspension for the code of some nonatomic expression e (referenced by the pointer p_e) and pushes a pointer to the suspension onto stack W. If the expression is an application, then the environment is represented by frame pointers p_A and p_T found in the topmost positions on stacks A and T, respectively; if it is an abstraction, then the environment included is just a tilde frame pointer p_T (with a *nil* value replacing the frame pointer p_A). In all other cases, the pointer returned by the instruction just points to the code of the expression or to an evaluated graph node.

COPY_AW i copies the i-th entry (a pointer) of the topmost A-frame onto the top of stack W.

COPY_TW i copies the i-th entry (a pointer) of the topmost T-frame onto the top of stack W.

MKFRAME_A n takes the topmost n entries off stack W, puts them in a frame allocated in the heap, and pushes a pointer to this frame onto stack A.

MKFRAME_T n takes the topmost n entries off stack W, puts them in a frame allocated in the heap, and pushes a pointer to this frame onto stack T.

FREE_A pops the topmost frame pointer off stack A.

FREE_T pops the topmost frame pointer off stack T.

BRA p_f realizes a function call: it saves as return continuation on R the remainder of the code in C, copies into C instead the function code referenced by the pointer p_f and decrements the count value rr by one.

JFALSE p_{ff} realizes a conditional branch: it inspects the topmost value of stack W and, if this value is found to be `false`, copies the code referenced by

the pointer p_{ff} in front of C; otherwise code execution continues with the next instruction in sequence.

RET effects a return from a function call in that it restores in C a return continuation retrieved from R.

REDUCE i forces the reduction of a suspension referenced by the i-th pointer entry relative to the top of stack W by prepending the suspension's code to C and pushing the pointers to the suspension's environment frames onto stacks A and T; if the i-th entry points to an evaluated expression, then the instruction has no effect other than pushing this pointer on top of W.

UPDATE i overwrites the graph node referenced by the pointer found in the i-th position relative to the top of W with the graph referenced by the topmost pointer in W.

RTT returns from the execution of a REDUCE instruction by popping the topmost frame pointers (or *nil* entries) off the stacks A and T.

AP n is the most complex instruction of the set: it attempts to apply the top element of W to the n elements underneath. If the top element is a primitive function or a pointer to function code whose arity is less than or equal to n, then the application is actually reduced (and the count value rr is decremented); otherwise a closure is constructed in the heap. In either case, the function and at most as many arguments as the function requires are popped off the stack, and (a pointer to) the result is pushed instead. If the function consumes fewer than n arguments, then another attempt is made to apply the result to the remaining arguments.

ENTRY is the first instruction of a program; it initializes the runtime structures of the machine.

EXIT is the last instruction of a program that terminates its execution.

In addition, there are a number of parameterless arithmetic, logic, and relational instructions such as ADD, SUB, ..., CMP, LT, ... that take two arguments off stack W and push a result instead.

A formal definition of these instructions in terms of state transition rules is given in Fig. 10.5, using the same notation for operations involving the heap as in Sect. 8.2.[6]

To keep these definitions as concise as possible, we have made descriptors for things held in the heap explicit only where really needed to precisely specify what is going on, which are the descriptors for suspensions (or closures), but have dropped the descriptors of expressions (including abstractions) and frames, and instead used direct pointers to these objects.

So, we use the following notation:

- $pp \rightsquigarrow [\ p_A\ p_T \mid p_e\ ]$ for the descriptor of a suspension (or closure) referenced by pp, which in turn includes pointers p_A and p_T to the A- and T-frames,

[6] Of the primitive instructions, only ADD has been defined, as all the other instructions basically work in the same way.

$$(\text{PUSH\_C } item : C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ p_c : W,\ T,\ R\ ),\ p_c \rightsquigarrow item : H,\ rr\ )$$

$$(\text{PUSH\_P } pp : C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ pp : W,\ T,\ R\ ),\ H,\ rr\ )$$

$$(\text{PUSH\_FUN } prf : C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ prf : W,\ T,\ R\ ),\ H,\ rr\ )$$

$$(\text{MKSUSP } p_e : C,\ (\ p_A : A,\ W,\ p_T : T,\ R\ ),\ H[\ p_e \rightsquigarrow code[e]\ ],\ rr\ ) \mid e = (\ e_0\ e_1 \ldots e_n\ ) \rightarrow (\ C,\ (\ p_A : A,\ pp : W,\ p_T : T,\ R\ ),\ pp \rightsquigarrow [\ p_A\ p_T \mid p_e\ ] : H[],\ rr\ )$$

$$(\text{MKSUSP } p_e : C,\ (\ A,\ W,\ p_T : T,\ R\ ),\ H[\ p_e \rightarrow code[e]\ ],\ rr\ ) \mid e = \Lambda v.e' \rightarrow (\ C,\ (\ A,\ pp : W,\ p_T : T,\ R\ ),\ pp \rightarrow [\ nil\ p_T \mid p_e\ ] : H[],\ rr\ )$$

$$(\text{MKSUSP } p_e : C,\ (\ nil,\ W,\ nil,\ R\ ),\ H[\ p_e \rightsquigarrow code[e]\ ],\ rr\ ) \rightarrow (\ C,\ (\ nil,\ pp : W,\ nil,\ R\ ), pp \rightsquigarrow [\ nil\ nil \mid p_e\ ] : H[],\ rr\ )$$

$$(\text{COPY\_AW } i : C,\ (\ p_A : A,\ W,\ T,\ R\ ),\ H[\ p_A \rightsquigarrow < p_1 \ldots p_i \ldots p_n >\ ],\ rr\ ) \rightarrow (\ C,\ (\ p_A : A,\ p_i : W,\ T,\ R\ ), H[],\ rr\ )$$

$$(\text{COPY\_TW } i : C,\ (\ A,\ W,\ p_T : T,\ R\ ),\ H[\ p_T \rightsquigarrow < p_1 \ldots p_i \ldots p_n >\ ],\ rr\ ) \rightarrow (\ C,\ (\ A,\ p_i : W,\ p_T : T,\ R\ ), H[],\ rr\ )$$

$$(\text{MKFRAME\_A } n : C,\ (\ A,\ p_1 : \ldots p_n : W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ p_A : A,\ W,\ T,\ R\ ), p_A \rightsquigarrow < p_1 \ldots p_n > : H,\ rr\ )$$

$$(\text{MKFRAME\_T } n : C,\ (\ A,\ p_1 : \ldots p_n : W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ W,\ p_T : T,\ R\ ), p_T \rightsquigarrow < p_1 \ldots p_n > : H,\ rr\ )$$

$$(\text{FREE\_A: } C,\ (\ p_A : A,\ W,\ T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ )$$

$$(\text{FREE\_T: } C,\ (\ A,\ W,\ p_T : T,\ R\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ )$$

$$(\text{JFALSE } p_{ff} : C,\ (\ A,\ p_{cc} : W,\ T,\ R\ ),\ H[\ p_{ff} \rightsquigarrow code[e], p_{cc} \rightsquigarrow false\ ],\ rr\ ) \rightarrow (\ code[e],\ (\ A,\ W,\ T,\ R\ ),\ H[],\ rr\ )$$

$$(\text{JFALSE } p_{ff} : C,\ (\ A,\ p_{cc} : W,\ T,\ R\ ),\ H[\ p_{ff} \rightsquigarrow code[e], p_{cc} \rightsquigarrow true\ ],\ rr\ ) \rightarrow (\ C,\ (\ A,\ W,\ T,\ R\ ),\ H[],\ rr\ )$$

$$(\text{BRA } p_f : C,\ (\ A,\ W,\ T,\ R\ ),\ H[\ p_f \rightsquigarrow code[e]\ ],\ rr\ ) \mid rr > 0 \rightarrow (\ code[e] : nil,\ (\ A,\ W,\ T,\ (\ C,\ R\ )\ ),\ H[],\ rr - 1\ )$$

$$(\text{RET: } nil,\ (\ A,\ W,\ T,\ (\ C,\ R\ )\ ),\ H,\ rr\ ) \rightarrow (\ C,\ (\ A,\ W,\ T,\ R\ ),\ H,\ rr\ )$$

Fig. 10.5. A formal definition of the LASM instruction set (continued on the next page)

$$(\text{REDUCE } i : C,\ (A,\ p_0 : \ldots : p_i : W,\ T,\ R),\ H[\, p_i \rightsquigarrow [\, p_A\ p_T \mid p\,], p \rightsquigarrow code[e]\,],\ rr\ ) \rightarrow (code[e] :\text{RTT: } C,\ (p_A : A,\ p_0 : \ldots : p_i : W,\ p_T : T,\ R),\ H[],\ rr\ )$$

$$(\text{REDUCE } i : C,\ (A,\ p_0 : \ldots : p_i : W,\ T,\ R),\ H[\, p_i \rightsquigarrow value\,],\ rr\ ) \rightarrow (C,\ (A,\ p_i : p_0 : \ldots : p_i : W,\ T,\ R),\ H[],\ rr\ )$$

$$(\text{UPDATE } i : C,\ (A,\ p_{res} : \ldots : p_i : W,\ T,\ R),\ H[\, p_{res} \rightsquigarrow val,\ p_i \rightsquigarrow \ldots\,],\ rr\ ) \rightarrow (C,\ (A,\ \ldots : p_i : W,\ T,\ R),\ H[\, p_{res} \rightsquigarrow val, p_i \rightsquigarrow val\,],\ rr\ )$$

$$(\text{RTT} : C,\ (p_A : A,\ W,\ p_T : T,\ R),\ H,\ rr\ ) \rightarrow (C,\ (A,\ W,\ T,\ R),\ H,\ rr\ )$$

$$(\text{AP } n : C,\ (A,\ p_f : W,\ T,\ R),\ H[\, p_f \rightsquigarrow code[fun]_n\,],\ rr\ ) \mid (rr > 0\ ) \rightarrow (code[fun]_n : nil,\ (A,\ W,\ T,\ (C,\ R\ )\ ),\ H[],\ rr - 1\ )$$

$$(\text{AP } n : C,\ (A,\ p_f : W,\ T,\ R),\ H[\, p_f \rightsquigarrow code[fun]_k\,],\ rr\ ) \mid (k < n\ ) \wedge (rr > 0\ ) \rightarrow (code[fun]_k : nil,\ (A,\ W,\ T,\ (\ \text{AP } n - k : C,\ R\ )\ ),\ H[],\ rr - 1\ )$$

$$(\text{AP } n : C,\ (A,\ p_f : p_1 : \ldots : p_n : nil,\ p_T : T,\ R),\ H[\, p_f \rightsquigarrow code[fun]_k\,],\ rr\ ) \mid\ (k > n\ ) \rightarrow (C,\ (A,\ p_{clos} : W,\ p_T : T,\ \ R\ ),\ p_{clos} \rightsquigarrow [\, p_A\ p_T \mid p_f\,],\ p_A \rightsquigarrow < p_1 \ldots p_n > : H,\ rr\ )$$

$$(\text{ADD} : C,\ (A,\ p_1 : p_2 : W,\ T,\ R),\ H[\, p_1 \rightsquigarrow num_1, p_2 \rightsquigarrow num_2\,],\ rr\ ) \rightarrow (C,\ (A,\ p_{res} : W,\ T,\ R),\ p_{res} \rightsquigarrow (num_2 + num_1) : H[],\ rr\ )$$

Fig. 10.5. A formal definition of the LASM instruction set (continued from previous page)

respectively, that make up the environment for the code to which p_e points. If this code happens to be that of an abstraction, then the environment consists only of a T-frame, i.e., the pointer p_A is set to nil.

- $p_{A|T} \rightsquigarrow < p_1 \ldots p_n >$ for an A- or T-frame of n pointers (to suspensions), to which $p_{A|T}$ points.
- $p_e \rightsquigarrow code[e]$ for the code of an expression e referenced by a pointer p_e.
- $p_f \rightsquigarrow code[fun]_k$ for the code of a function (abstraction) fun of arity k to which p_f points.

Reference counting is omitted in these rules as heap space, for the sake of defining the state transitions effected by the instructions, may be considered unlimited.

10.3.2 Compilation to LASM Code *

Compilation of source programs to LASM code may be defined by a compilation scheme similar to that of the G-machine. It specifies a set of rules that

translates syntactical forms of the source language into pieces of target code. We take as the source language a kernel of AL that has been preprocessed as described in Sect. 10.1. Its syntax is given by

$$\begin{aligned} e =_s\ & const \mid pp \mid prf \mid \#i \mid\ \sim j \mid \\ & \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\ & (\ e_0\ e_1\ \ldots\ e_r\ ) \mid\ (\sim\tilde{\Lambda}_{v_1} \ldots \tilde{\Lambda}_{v_s}.e_t\ e_1\ \ldots\ e_s\ ) \mid \\ & \Lambda_{u_1} \ldots \Lambda_{u_n}.e \mid \\ & \texttt{letrec}\ f_1 = e_1 \ldots f_m = e_m\ \texttt{in}\ e_0\ \ , \\ e_t =_s\ & \Lambda_{u_1} \ldots \Lambda_{u_n}.e \mid \texttt{letrec} \ldots \texttt{in}\ e_0 \end{aligned}$$

i.e., the expressions that we have to deal with are constant values, pointers pp to things that are held in the heap, primitive function symbols prf, A- and T-indices that replace bound-variable occurrences, conditionals, ordinary and tilde applications, ordinary abstractions and `letrec`s, respectively. We remember that by construction the operator expression of a tilde application is a tilde abstraction of matching arity, and that the body expression e_t of this abstraction is either an ordinary abstraction or a `letrec`.

The top-level compilation scheme $\mathcal{C}$ is defined as

$$\mathcal{C}[\ e : es\ ] \implies code[\ e\ ]; \mathcal{C}[\ es\ ]\ .$$

It splits the compiler input $e : es$ up into a head expression e that conforms to one of the above syntactical forms, and a tail es that represents the remainder of what needs to be compiled.[7] This approach generates a very orderly code structure that uses only branch instructions, e.g., to function code or to code for the components of conditionals, which are complemented by return instructions. Code referenced by pointers from within the code for e can thus be placed into the free space immediately following the code for es (and possibly other pieces of code that have, in the course of compiling e, been placed there), and compilation may be postponed until later.

A typical example is the compilation of `letrec`s embedded in larger contexts. It splits up into compiling first their body expressions, then the (unrelated) remaining tails that follow and finally the function definitions. Occurrences of `letrec`-bound variables f_i either in the body of the entire `letrec` or in one of the function bodies may be replaced by unique symbolic labels p_{fi} that remain undefined until the respective functions are compiled as well.

Following what we have said in the preceding section about suppressing descriptors whenever convenient, function definitions $f_i = \Lambda_{u_1} \ldots \Lambda_{u_r}.e_i$ are compiled as

$$p_{f_i}\ \rightsquigarrow\ code[e_i]_r\ \ ,$$

which turns the label p_{f_i} into a pointer to the location where $code[e_i]_r$ can be found in the heap, but we should keep in mind that it implies creation of a descriptor that introduces a level of indirection between the pointer p_{f_i} and the

[7] The symbol ';' catenates two pieces of code (or two consecutive instructions).

code that evaluates the body expression e_i. Similarly, when compiling `letrec`s, it is implicitly assumed that a graph structure such as that in Fig. 10.3 is created.

The complete compilation scheme $\mathcal{C}$ that specifies how each of the above syntactical forms translates into code is given in Fig. 10.6.

(1) $\mathcal{C}[\, const \;:\; es\,] \implies$ PUSH_C $const$; $\mathcal{C}[\, es\,]$;

(2) $\mathcal{C}[\, \#i \;:\; es\,] \implies$ COPY_AW i; $\mathcal{C}[\, es\,]$;

(3) $\mathcal{C}[\, \sim i \;:\; es\,] \implies$ COPY_TW i; $\mathcal{C}[\, es\,]$;

(4) $\mathcal{C}[\,$ **if** e_0 **then** e_1 **else** $e_2 \;:\; es\,] \implies \mathcal{C}[\, e_0\,]$; BRA p_{if}; $\mathcal{C}[\, es\,]$;
$p_{if} \rightsquigarrow$ JFALSE p_{false}; $\mathcal{C}[\, e_1\,]$; RET; $\quad p_{false} \rightsquigarrow \mathcal{C}[\, e_2\,]$; RET;

(5) $\mathcal{C}[\,(\, e_0\; e_1\; \ldots\; e_{r-1}\; e_r\,) \;:\; es\,] \implies \mathcal{L}[\, e_r : e_{r-1} : \ldots : e_1 : e_0 : ap^r : es\,]$;

(5a) $\mathcal{L}[\, e_0 : ap^r : es\,] \implies \mathcal{C}[\, e_0\,]$; AP r; $\mathcal{C}[\, es\,]$;

(5b) $\mathcal{L}[\, e \;:\; es\,] \implies$
$$\begin{cases} \text{MKSUSP } pp;\ \mathcal{L}[\, es\,];\ \ pp \rightsquigarrow \mathcal{C}[\, e\,];\ \text{RTT}; \\ \qquad \text{if } e = \Lambda.e' \mid (\, e'_0\; e'_1 \ldots) \mid\ < e'_1 \ldots e'_n > \\ \mathcal{C}[\, e\,];\ \mathcal{L}[\, es\,];\ \ \text{otherwise} \end{cases}$$

(6) $\mathcal{C}[\, \Lambda_{u_1}\; \ldots\; \Lambda_{u_r}.e \;:\; es\,] \implies$ PUSH_P p_f; $\mathcal{C}[\, es\,]$
$p_f \rightsquigarrow$ MKFRAME_A r;$\mathcal{C}[\, e\,]$; FREE_A; RET;

(7) $\mathcal{C}[\,$ **letrec** $f_1 = e_1 \ldots f_n = e_n$ **in** $e_0 \;:\; es\,] \implies \mathcal{C}[\, e_0\,]$; $\mathcal{C}[\, es\,]$;
$p_{f_1} \rightsquigarrow \mathcal{F}[\, e_1\,]$; $\ldots$; $p_{f_n} \rightsquigarrow \mathcal{F}[\, e_n\,]$;

(7a) $\mathcal{F}[\, \Lambda_{u_1}\; \ldots\; \Lambda_{u_r}.e\,] \implies$ MKFRAME_A r; $\mathcal{C}[\, e\,]$; FREE_A; RET;

(8) $\mathcal{C}[\,(\tilde{}\; e_0\; \ldots\; e_{s-1}\; e_s\,) \;:\; es\,] \implies \mathcal{C}[\, e_s\,]$; $\mathcal{C}[\, e_{s-1} : \ldots : e_0 : tap^s : es\,]$;

(8a) $\mathcal{C}[\, \tilde{\Lambda}_{v_1}\; \ldots\; \tilde{\Lambda}_{v_s}.e : tap^s : es\,] \implies$ BRA p_t; $\mathcal{C}[\, es\,]$;
$p_t \rightsquigarrow$ MKFRAME_T s; $\mathcal{C}[\, e\,]$; FREE_T; RET;

(9) $\mathcal{C}[\, prf\_1\,] \implies$ REDUCE 0; UPDATE 0; PUSH_FUN prf_1;

(10) $\mathcal{C}[\, prf\_2\,] \implies$ REDUCE 1; UPDATE 1;
REDUCE 0; UPDATE 0; PUSH_FUN prf_2;

Fig. 10.6. The LASM compilation rules

It may help to give an informal explanation of these compilation rules, and specifically of the codes that they generate:

- A constant value has a pointer to it pushed onto stack W (rule (1)).

- Occurrences of A- or T-indices translate into instructions that copy the pointers found under these index positions in the topmost A- or T-frames, respectively, and push them onto stack W (rules (2) and (3)).
- A conditional compiles to code that first computes the predicate expression e_0 and then branches to the code pointed to by p_{if}. The first instruction of this code checks the Boolean value computed by the code for e_0 and, if `false`, branches to the code for the alternative expression e_2; otherwise, it continues with the code of the consequent expression e_1. As both of these codes terminate with the instruction RET, control returns in either case to the instruction immediately following the branch instruction BRA p_{if} (rule (4)).
- An r-ary application calls for another compilation scheme $\mathcal{L}$ that is applied to its components in reverse order, followed by an apply symbol ap^r, followed by the remaining text (rule (5)). This rule prepares the compilation of nonatomic operand expressions for delayed evaluation by creating suspensions for them.
- The compilation scheme $\mathcal{L}$ breaks down into two rules, of which rule (5a) handles the special case of the last expression preceding the ap^r symbol – the operator of the application – which it compiles by switching back to $\mathcal{C}$, followed by an AP r instruction, followed by the code for the remaining text. Rule (5b) generates MKSUSP instructions for operand expressions that are applications, abstractions or lists, followed by $\mathcal{L}$-generated code of whatever remains to be compiled, followed by $\mathcal{C}$-compiled code for the operand itself. All other operands are directly compiled by calling $\mathcal{C}$ again.
- The code for an anonymous r-ary Λ-abstraction is called by a branch instruction and begins with the creation of an A-frame for r entries taken off stack W, followed by the code for the body expression, followed by instructions that release the A-frame again and return to the calling code (rule (6)). The code for a $\tilde{\Lambda}$-abstraction looks basically the same, except that it creates and releases a T-frame instead of an A-frame (rule (8a)).
- A top-level `letrec` compiles to code for its body expression and to codes for its individual function definitions (rule (7)). Since the function codes are already hidden behind pointers, their compilation requires another scheme $\mathcal{F}$ (rule (7a)) that avoids an additional level of indirection, but other than that is the same as rule (6).
- A tilde application compiles to code for its operands followed by a branch instruction that calls the code of the operator, which by construction can only be a tilde abstraction[8] (rule (8)).
- The codes for primitive unary and binary functions that are strict in their arguments, meaning that they must be normalized before the functions can

[8] Note that the operands of a tilde application, owing to the lifting mechanism that brings them about, can only be binding indices of either kind that straightforwardly compile to COPY_AW or COPY_TW instructions.

be applied, first force their evaluation by means of REDUCE instructions, followed by updates of the respective graph nodes (rules (9) and (10)).

The code for the entire program expression e is of the form

$$\text{ENTRY};\ \mathcal{C}[\![\, e \,]\!];\ \text{EXIT}\ ,$$

i.e., it is embedded in mandatory entry and exit instructions that initialize and terminate the machine, respectively.

10.3.3 Some Simple Code Optimizations

There are, of course, some opportunities for simple yet effective code optimizations. Primary targets are code fragments of the general form

$$\ldots \text{PUSH\_P}\ pp;\ \text{AP}\ r; \ldots \quad \text{or} \quad \ldots \text{PUSH\_FUN}\ prf;\ \text{AP}\ r; \ldots$$

that push either a pointer pp to an abstraction or a primitive function whose arity happens to be r, i.e., in both cases the arity matches the number of arguments to which the function is applied by the instruction AP r.

The first of these fragments may be replaced by an instruction BRA pp that calls the function code directly, i.e., without going through the superfluous motions of pushing its pointer onto the stack and then having it called by AP. Similarly, occurrences of the second code fragment may be replaced by dedicated primitive instructions such as ADD, MULT, ... GT, etc. that directly pop the topmost two entries off stack W and push the result of the operation instead. The AP instructions must be used only whenever it cannot be decided at compile time whether or not the arities match, which is generally the case if we have binding indices of either kind in operator position of applications.

Other targets for code optimizations are tail-recursive functions, as for instance in

$$\texttt{letrec}\ f = \Lambda_{u_1} \ldots \Lambda_{u_n}.(\ g\ a_1 \ldots a_m\ )\quad g = \Lambda_{v_1} \ldots \Lambda_{v_m}.(\ f\ b_1 \ldots b_n\ )\ \texttt{in}\ e_0\ .$$

Straightforward compilation by the rules of Fig. 10.6 would produce the codes

$$p_f\ \rightsquigarrow \text{MKFRAME\_A}\ n;\ \ldots\ ;\ \text{BRA}\ p_g;\ \text{FREE\_A};\ \text{RET};$$

and

$$p_g\ \rightsquigarrow \text{MKFRAME\_A}\ m;\ \ldots\ ;\ \text{BRA}\ p_f;\ \text{FREE\_A};\ \text{RET};$$

for the functions f and g, respectively (the dots stand for the codes that create suspensions for the operand expressions $a_1, \ldots,\ a_m$ and $b_1, \ldots,\ b_n$). Here we can do two things that simplify code execution. First, we can move the FREE_A instructions ahead of all instructions that neither change nor access the A-frame, which are at least the branch instructions BRA p_g and BRA p_f. That is to say, the A-frames can be released before the respective complementary functions are called. Thus, when these codes are called alternately, the

A-frames of the calling codes are released before the A-frames of the called codes are created, meaning that these tail calls can be executed in constant space. Second, since the codes of both functions end with BRA p_g; RET and BRA p_f; RET, they may be replaced by instructions JTAIL p_g and JTAIL p_f, respectively, that jump directly back to the beginning of the function codes. The JTAIL instructions, in contrast to the branch instructions, spare the machine the motions of saving and retrieving return continuations on stack R.

The tail-optimized versions of the above function codes thus are

$$p_f \rightsquigarrow \text{MKFRAME\_A } n;\ \ldots\ ;\ \text{FREE\_A; JTAIL } p_g;$$

and

$$p_g \rightsquigarrow \text{MKFRAME\_A } m;\ \ldots\ ;\ \text{FREE\_A; JTAIL } p_f;\quad .$$

Whenever a conditional makes up the entire body of an abstraction, as for instance in

$$h\ =\ \Lambda_{u_1} \ldots \Lambda_{u_m}.\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\quad ,$$

the code can be flattened to

$$\begin{array}{l} \mathcal{C}[\,e_0\,];\ \text{JFALSE } p_{false};\ \mathcal{C}[\,e_1\,];\ \ \text{FREE\_A } m;\ \ \text{RET};\\ \quad p_{false} \rightsquigarrow\ \mathcal{C}[\,e_2\,];\ \text{FREE\_A } m;\ \text{RET};\quad , \end{array}$$

which saves the BRA instruction that calls this piece of code and the complementary RET instruction. This rule may be recursively applied to nested conditionals.

Code efficiency may also be improved by eliminating, whenever possible, superfluous REDUCE and UPDATE instructions. This becomes possible if we have full applications of primitive unary or binary functions that are strict in their arguments, say, $(+\ e_1\ e_2)$. This applications may, in a first step, be schematically compiled to

$$\ldots;\ \mathcal{C}[\,e_2\,];\ \mathcal{C}[\,e_1\,];\ \ \text{REDUCE } 1;\ \text{UPDATE } 1;\ \text{REDUCE } 0;\ \ \text{UPDATE } 0;\ \text{ADD};\ldots\quad .$$

If the compilation of the operand expression e_1 or e_2 returns something other than instructions COPY_AW $\#i$ or COPY_TW $\#j$ (which retrieve suspensions from the environment), then the respective REDUCE and UPDATE instructions that force their evaluation may safely be discarded. A typical examples of such optimizations is the code for the expression (+ 43 (+ #2 123)):

$$\begin{array}{l} \ldots\text{PUSH\_C } 123;\ \text{COPY\_AW } \#2;\ \text{REDUCE } 1;\ \ \text{UPDATE } 1;\\ \quad \text{REDUCE } 0;\ \ \text{UPDATE } 0;\ \ \text{ADD};\\ \qquad \text{PUSH\_C } 43;\ \text{REDUCE } 1;\ \ \text{UPDATE } 1;\ \text{REDUCE } 0;\ \ \text{UPDATE } 0;\ \text{ADD};\ \ldots\\ \qquad\quad \rightarrow \ldots\text{PUSH\_C } 123;\ \text{COPY\_AW } \#2;\ \text{REDUCE } 0;\ \ \text{UPDATE } 0;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ADD};\ \text{PUSH\_C } 43;\ \text{ADD};\ldots\quad . \end{array}$$

It includes the interesting case of an inner application that contains a COPY_AW instruction whose corresponding REDUCEs and UPDATEs cannot be

deleted since at compile time it is not known whether or not a suspension must be evaluated as the second argument. However, both instructions may be removed from the outer application since the evaluation of the inner application is bound to return a numerical value in either case.

Room for more optimizations, particularly for such standard techniques as function inlining or loop unrolling, is somewhat limited insofar as a one-to-one correspondence between code-controlled machine-level graph reductions and equivalent high-level program transformations must be rigorously adhered to in order to be able to decompile intermediate states of code execution correctly into source language output.

Compiling the preprocessed version of the example program given in Sect. 10.1 yields the code shown in Fig. 10.7.

$p_e \rightsquigarrow$ ENTRY; PUSH_C 2; PUSH_C 1; BRA p_f; EXIT;

$p\_f \rightsquigarrow$ MKFRAME_T 2; COPY_TW 0; COPY_TW 1; BRA p_g; FREE_T; RET;

$p\_g \rightsquigarrow$ MKFRAME_A 2; COPY_AW 0; COPY_TW 0;
REDUCE 1; UPDATE 1; RTT; REDUCE 0; UPDATE 0; RTT; GT;
JFALSE p_m; COPY_AW 1; FREE_A; MKSUSP p_c1; JTAIL p_g;

$p\_m \rightsquigarrow$ MKSUSP p_c2; FREE_A; COPY_TW 1; BRA p_f; RET;

$p\_c1 \rightsquigarrow$ COPY_TW 0; PUSH_C 1; REDUCE 1; UPDATE 1; SUB; RTT;

$p\_c2 \rightsquigarrow$ COPY_AW 0; REDUCE 1; UPDATE 1; PUSH_C 1; PLUS; RTT;

Fig. 10.7. Optimized LASM code for the example program of Fig. 10.1

The top line of this code implements the tilde application ($\sim f\ 1\ 2$) that is the body expression of the outer `letrec`. Having pushed the two arguments onto W, the instruction BRA p_f calls the code for f that is shown in the second line. It computes the body expression ($g\ \sim 1\ \sim 0$) of the inner `letrec` in that it creates a T-frame from the two entries in W, moves these two entries in reverse order from T back to W and then branches to the code for the function g in the third line (the T-frame is released again after returning from g). This code first creates an A-frame for the two entries taken off the top of W, then evaluates the predicate of the `if_then_else` clause and, depending on the outcome, either branches via the instruction JFALSE to the alternative code to which p_m points (fourth line from the top) or continues in line with the consequent code. Both pieces of code include a MKSUSP instruction for applications that are arguments of abstractions (these are ($-\ 1\ \sim 0$) as argument for g and ($+\ 1\ \#0$) as argument for f) since under a lazy regime their evaluation must be suspended until a demand arises later on. This happens whenever these suspensions are substituted for occurrences of binding

indices of either kind in operand positions of strict (primitive) functions such as $+$, $-$, $*$, `gt`, ... etc. This is the case for the instruction GT in the predicate code of the function g that is preceded by two pairs of REDUCE, UPDATE instructions to force the evaluation of the arguments pushed onto stack W by the instructions COPY_AW 0 and COPY_TW 0. Likewise, one such pair precedes the instructions SUB and PLUS in the suspension codes referenced by the pointers p_c1 and p_c2, respectively (the two lines at the bottom of the figure). The compiler must generate these sequences as a precautionary measure since it has no direct way of knowing what kind of argument expressions are being substituted.

10.4 The Strict Abstract Stack Machine SASM

The strict variant SASM of the π–RED machinery shares with the lazy variant LASM essentially the same runtime structures and the very basic operating principles. However, the particularities of using these structures, the instruction set and also compilation to SASM code are quite different, and to some extent also simpler and more direct. The single most important reason for this is that under a strict (or applicative order) regime the operands of applications are, at some risk of getting trapped in runaway recursions, directly evaluated before the operators are applied, as opposed to suspending their evaluation until a demand arises, as under a lazy regime. As a consequence, operand expressions need not be embedded in suspensions that hold on to their environments, i.e., to A- and T- frame pointers, beyond the lifetimes of the function calls or tilde applications that bring them about. Instead, A- and T-frames may be released in reverse order of their creation immediately after control returns from the respective abstraction codes. This last-in-first-out (LIFO) order suggests that the frames rather than pointers to them should be directly pushed onto the A- and T-stacks, thus saving one level of indirection, which significantly speeds up environment accesses. Garbage collection also becomes decidedly simpler and less frequent.

Moreover, keeping the environment frames directly in the stacks suggests an elegant solution to the problem of making an argument frame that has built up in the working stack W available in stack A. Rather than explicitly moving it by an instruction similar to MKFRAME_A, as the LASM does, we can simply flip, upon a function call, the argument stack A with the workspace stack W, without moving a single entry. The arguments that have been pushed onto W by the calling code thus become directly accessible in what, after the stack switch, becomes stack A of the called function code. The function value computed by the called code is put on top of its stack W. Immediately before returning from a function call, (the pointer to) this value must therefore be moved from W to A to make sure that, after another stack switch that restores the original stack configuration, this value ends up on top of stack W of the calling code.

The same trick can, of course, also be played between stacks W and T: the arguments of a tilde application are stacked up in W, entering the code of the tilde abstractions effects a stack switch between W and T that renders the arguments accessible in what then becomes stack T, and returning from this code restores the original stack configuration.

Again, the complete environment for executing a piece of code is thus available in what are the topmost A- and T-frames that now are held directly in the stacks. Accesses to the frames are specified by fixed offsets relative to the stack tops and, as in the LASM, directly derive from the A- and T-indices generated by the preprocessor. Switching the stacks also enables the SASM compiler to generate code that releases as early as possible and right from the top of the stacks the A- or T-frames that are no longer needed, thus minimizing the consumption of stack and heap space.

10.4.1 The SASM Instruction Set

In comparison with the LASM, the SASM needs fewer and less complex instructions. The instructions that are dropped are essentially those that deal with the creation and handling of suspensions, i.e., MKSUSP, REDUCE and UPDATE, and the MKFRAME_x instructions. The effects of the latter are now realized by simple stack switches performed as part of the branch instructions that call abstraction codes, and as part of the complementary return instructions. However, as a consequence, we now need three variants of branch and return instructions to distinguish between the stacks that need to be flipped and those that require no flips at all.

Thus, the SASM instruction set includes the following:

PUSH_W *item* pushes onto stack W an item that may be either a constant value, a primitive function symbol, or a pointer to some piece of graph.

COPY_AW i copies the i-th entry relative to the top of stack A and pushes it on top of stack W.

COPY_TW i copies the i-th entry relative to the top of stack T and pushes it on top of stack W.

FREE_A n pops the topmost n entries off stack A.

FREE_T n pops the topmost n entries off stack T.

BRA_F p_f branches to the code of an ordinary abstraction: it saves the code that is left over in C as return continuation in stack R, copies into C the abstraction code to which p_f points, flips the stacks A and W, and decrements the count value rr by one.

BRA_T p_t branches to the code of a tilde abstraction: it does the same as BRA_F except that it flips stacks T and W rather than A and W.

BRA_C p_c branches to the code of a conditional and saves the code that is left over in C as the return continuation in R; no stacks are flipped.

JFALSE p_{if} is exactly as in the LASM: it inspects the topmost value on stack W and, if it is `false`, copies the code to which p_{if} points in front of C; otherwise code execution continues with the next instruction in sequence.

RTF is used to return from ordinary function code: the instruction moves the function value from stack W to stack A, then flips the two stacks to restore the stack configuration as before the function call, and finally retrieves the return continuation from R to restore it in C.

RTT is used to return from reducing a tilde application: it does exactly the same as RTF except that it flips stacks T and W.

RTC is used to return from a conditional simply by restoring in C a return continuation retrieved from R; no stacks are flipped.

AP n works like the equivalent LASM instruction: it attempts to apply the top element of W to the n elements that are underneath. If the top element is a pointer to abstraction code whose arity is less than or equal to n, then stacks A and W are flipped and the code is actually executed (and rr is decremented). If the top element is a primitive function of matching arity, then it is applied directly to the topmost n entries in W that, together with the function symbol itself, are subsequently popped, and the result value is pushed instead. If the parameter n of the instruction is less than the arity of the function, then a closure is formed, and the pointer to it is pushed onto W; no stacks are flipped.[9]

Again, this set is complemented by parameterless instructions such as ADD, SUB, ..., CMP, ... that take two arguments off stack W and push a result value instead.

10.4.2 Compilation to SASM Code *

Compiling preprocessed AL programs to SASM code follows essentially the same rules as given in Fig. 10.6 for LASM. The SASM rules are summarized in Fig. 10.8.

Here we have again a top-level compilation scheme

$$\mathcal{C}[\, e : es \,] \implies \mathcal{C}[\, e \,];\ \mathcal{C}[\, es \,] \ ,$$

which applies to a head expression e and recursively to some tail es. With one minor exception (rule (7a)) it does the entire job.

The primary difference from the LASM compiler relates to the compilation of ordinary applications (rule (5)). The operand expressions are directly compiled by $\mathcal{C}$ to code that effects their strict evaluation, rather than creating suspensions. Other differences relate to the generation of different branch instructions and the complementary return instructions for conditionals (rule (4)) and for ordinary and tilde abstractions (rules (5a) and (8a)) that effect the respective stack switches, to dropping the MKFRAME_X instructions from the abstraction codes, and to adding to the FREE_X instructions parameters

[9] At this point we need to remember that tilde applications are always full applications, i.e., calling the codes of tilde abstractions need never involve AP instructions as it can always be directly handled by BRA_T instructions.

that specify the number of stack entries that must be popped (rules (5a) and (8a) again, and also rules (6) and (7a)). The remaining compilation rules are exactly the same as for the LASM.

(1) $\mathcal{C}[\, item \;:\; es \,] \implies$ PUSH_W $item$; $\mathcal{C}[\, es \,]$;

(2) $\mathcal{C}[\, \#i \;:\; es \,] \implies$ COPY_AW i; $\mathcal{C}[\, es \,]$;

(3) $\mathcal{C}[\, \sim i \;:\; es \,] \implies$ COPY_TW i; $\mathcal{C}[\, es \,]$;

(4) $\mathcal{C}[\, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \;:\; es \,] \implies \mathcal{C}[\, e_0 \,]$; BRA_C p_{if}; $\mathcal{C}[\, es \,]$;
$p_{if} \rightsquigarrow$ JFALSE p_{false}; $\mathcal{C}[\, e_1 \,]$; RTC; $\quad p_{false} \rightsquigarrow \mathcal{C}[\, e_2 \,]$; RTC;

(5) $\mathcal{C}[\, (\ e_0\ e_1\ \ldots\ e_{r-1}\ e_r\ )\ :\ es \,] \implies \mathcal{C}[\, e_r \,] : \mathcal{C}[\, e_{r-1} : \ \ldots\ : e_1 : e_0 : ap^r : es \,]$;

(5a) $\mathcal{C}[\, \Lambda_{u_1}\ \ldots\ \Lambda_{u_r}.e : ap^r : es \,] \implies$ BRA_F p_f; $\mathcal{C}[\, es \,]$;
$p_f \rightsquigarrow \mathcal{C}[\, e \,]$; FREE_A r; RTF;

(5b) $\mathcal{C}[\, ap^r : es \,] \implies$ AP r; $\mathcal{C}[\, es \,]$;

(6) $\mathcal{C}[\, \Lambda_{u_1}\ \ldots\ \Lambda_{u_r}.e \;:\; es \,] \implies$ PUSH_W p_f; $\mathcal{C}[\, es \,]$;
$p_f \rightsquigarrow \mathcal{C}[\, e \,]$; FREE_A r; RTF;

(7) $\mathcal{C}[\, \texttt{letrec}\ f_1 = e_1\ \ldots\ f_n = e_n\ \texttt{in}\ e_0 \;:\; es \,] \implies \mathcal{C}[\, e_0 \,]$; $\mathcal{C}[\, es \,]$;
$p_{f_1} \rightsquigarrow \mathcal{F}[\, e_1 \,]$; $\ldots$; $p_{f_n} \rightsquigarrow \mathcal{F}[\, e_n \,]$;

(7a) $\mathcal{F}[\, \Lambda_{u_1}\ \ldots\ \Lambda_{u_r}.e \,] \implies \mathcal{C}[\, e \,]$; FREE_A r; RTF;

(8) $\mathcal{C}[\, (\sim e_0\ e_1\ \ldots\ e_{s-1}\ e_s\ )\ :\ es \,] \implies \mathcal{C}[\, e_s \,]$; $\mathcal{C}[\, e_{s-1} : \ \ldots\ : e_0 : tap^s : es \,]$

(8a) $\mathcal{C}[\, \tilde{\Lambda}_{v_1}\ \ldots\ \tilde{\Lambda}_{v_s}.e : tap^s : es \,] \implies$ BRA_T p_t; $\mathcal{C}[\, es \,]$;
$p_t \rightsquigarrow \mathcal{C}[\, e \,]$; FREE_T s; RTT;

Fig. 10.8. The SASM compilation rules

There are again a few straightforward code optimizations that are similar to those discussed in Sect. 10.3.3 for the LASM.

Eliminating AP r instructions whenever the parameter r equals the arity of the abstraction in operator position has already been taken care of by rule (5a). The same can be done if the operator is a primitive function of matching arity, in which case the code fragment $\ldots$ PUSH_W prf; AP r; can be replaced by a dedicated parameterless instruction such as ADD, MULT, $\ldots$ that takes its arguments off the workspace stack W and pushes a result value instead.

Tail-recursive functions such as in

$$\texttt{letrec}\ f = \Lambda_{u_1} \ldots \Lambda_{u_n}.(g\ a_1 \ldots a_m)\ \ g = \Lambda_{v_1} \ldots \Lambda_{v_m}.(f\ b_1 \ldots b_n)\ \texttt{in}\ e_0$$

compile in a first step to the codes

$$p_f \rightsquigarrow \mathcal{C}[\, a_m \,]; \ldots ; \mathcal{C}[\, a_1 \,]; \text{BRA\_G } p_f; \text{ FREE\_A } m; \text{ RTF};$$

and

$$p_g \rightsquigarrow \mathcal{C}[\, b_n \,]; \ldots ; \mathcal{C}[\, b_1 \,]; \text{BRA\_G } p_g; \text{ FREE\_A } n; \text{ RTF};$$

for the functions f and g. Next, the FREE_A instructions can again be moved ahead of all instructions that do not touch the topmost A-frame, which in this particular case are at least the branch instructions preceding them, and the branch | return instruction combinations can subsequently be replaced by JTAIL instructions that in fact turn the tail-recursive function calls into mutually entangled iteration loops. We thus get:

$$p_f \rightsquigarrow \mathcal{C}[\, a_m \,]; \ldots ; \mathcal{C}[\, a_1 \,]; \text{ FREE\_A } m; \text{ JTAIL } p_f;$$

and

$$p_g \rightsquigarrow \mathcal{C}[\, b_n \,]; \ldots ; \mathcal{C}[\, b_1 \,]; \text{ FREE\_A } n; \text{ JTAIL } p_g; \quad .$$

These tail jumps cause a small problem, though. They must of course switch the stacks A and W as the BRA_F instruction does in order for the code to always find the right things in the right stacks. To restore the correct stack configuration upon returning from a sequence of tail jumps (which our example code never does since the functions do not include conditionals that will eventually terminate them), the return continuation stacked up on R when calling one of the functions from somewhere else must include a single tail flag. Starting with the initial value zero, this flag must be flipped with every tail jump. When this flag is set to one upon executing the complementary RTF instruction, the stacks remain as they are; otherwise they must be flipped again.

Conditionals that make up an entire abstraction body may be flattened in exactly the same way as described in Sect. 10.3.3.

Compilation of the example program of Sect. 10.1, including all optimizations, yields the SASM code shown in Fig. 10.9. In comparison with the equivalent LASM code it looks decidedly simpler and more concise since it directly evaluates arguments of function applications rather than going through the motions of wrapping them up in suspensions and forcing their evaluation later on and in other contexts.

Again we have a top line of code for the application (f 1 2) of the outermost `letrec` that calls the code of the function f to which p_f points, thereby switching stacks W and T. This code moves from what is now stack T back to stack W the two arguments with which the function g must be called (these two arguments are removed from T immediately after control returns from g). The code of g is entered through the pointer p_g (third line from the top), takes one argument from each of the stacks A and T, evaluates the predicate of the conditional and, if `true`, continues with the consequent code following the JFALSE instruction. The JTAIL instruction at the end of this piece of code

$p_e \rightsquigarrow$ ENTRY; PUSH_W 2; PUSH_W 1; BRA_T p_f; EXIT;

$p_f \rightsquigarrow$ COPY_TW 0; COPY_TW 1; BRA_F p_g; FREE_T 2; RTT;

$p_g \rightsquigarrow$ COPY_AW 0; COPY_TW 0; GT;
JFALSE p_m; COPY_AW 1; FREE_A 2; COPY_TW 0; PUSH_W 1; MINUS; JTAIL p_g;

$p_m \rightsquigarrow$ COPY_AW 0; FREE_A 2; PUSH_W 1; ADD; COPY_TW 1; BRA_T p_f; RTF;

Fig. 10.9. Optimized SASM code for the example program of Fig. 10.1

returns control back to its beginning. If the predicate evaluates to `false`, the code of the alternative is entered through the pointer p_m, which calls the code of f again through the BRA_T instruction. This instruction, together with the RTF instruction that follows next, cannot be replaced by a tail jump since, for good reasons, there is no optimizing rule to this effect.

10.4.3 Code Execution

Code execution in the SASM starts with the count value rr set to some nonzero value that specifies an upper limit on the number of function calls to be performed, with the initial program graph held in the heap, and with the top line of the code loaded into the structure C. By convention, the first instruction of this piece of code is ENTRY, which clears the runtime stacks. Orderly termination with the EXIT instruction at the end of this code leaves the stacks A, T, R empty and a single entry representing the result of the computation in the workspace stack W. This entry is either a pointer to a coherent graph or a basic value.

Intermediate states of code execution have the stacks filled according to the nesting of tilde applications and of function calls (applications of ordinary Λ-abstractions). The entries in stacks A, T, W are basic values, primitive function symbols and pointers to graph fragments held in the heap, e.g., closures created by partial applications. The entries in R are code fragments that represent return continuations of function calls.[10]

If the count value rr comes down to zero in some intermediate state, the machine must nevertheless somehow continue executing the remaining code to terminate in an orderly manner, i.e., with a pointer to a coherent graph representing a partially reduced expression as the sole entry in W. The remaining code specifies, in the form of RTx instructions, the complete return path to the terminal state, and the FREE_x instructions encountered along this path clear the stacks. These instructions must be executed exactly as defined. However, all function calls and tilde applications must, from the

[10] In an implementation of the machine, stack R would contain return addresses pointing to the first instruction of rather than to the full return continuation.

point at which rr is exhausted forward, be treated as irreducible and, as is standard procedure in such cases, be wrapped up in closures.

When executing an instruction AP r with $rr = 0$, the machine inspects the topmost entry of stack W. If this entry is a pointer p_f to an ordinary abstraction, then the instruction takes its parameter r and retrieves from the abstraction's descriptor the parameter s (which is the number of relatively free variables that have been abstracted out) to create in the heap a closure that we choose to represent here in a more convenient form as

$$p_{clos} \rightsquigarrow (\sim (p_f\ a_1 \ldots a_r)\ b_1 \ldots b_s)$$

rather than as a heap structure

$$p_{clos} \rightsquigarrow [\ p_a\ p_t \mid p_f\ ] : p_a \rightsquigarrow < a_1 \ldots a_r > : p_t \rightsquigarrow < b_1 \ldots b_s > : H \quad ,$$

which, similarly to the notation that we have used to describe the LASM in Sect. 10.3, makes the descriptor explicit. Here p_f and $a_1 \ldots a_r$ are the topmost $r+1$ entries of stack W, which must be popped, and $b_1 \ldots b_s$ are the topmost s entries in stack T, which must be copied since they may still be needed elsewhere. This closure represents the application in unevaluated form.

If the top item on W is something other than a pointer to function code, say a constant value or a primitive function symbol prf, then the closure is of the simpler form

$$p_{clos} \rightsquigarrow (prf\ a_1 \ldots a_r) \quad ,$$

i.e., it is just made up from the entries that need to be cleared off the workspace stack.

The instruction BRA_F p_f creates the same closure as does AP r, except that only r items must be taken off W, and that both the indices r and s that determine the number of arguments to be included must be retrieved from the function descriptor.

The instruction BRA_T p_f looks up the arity index s in the descriptor of the tilde abstraction and creates a closure

$$p_{clos} \rightsquigarrow (\sim p_f\ b_1 \ldots b_s) \quad ,$$

with $b_1 \ldots b_s$ popped off stack T.

The closure pointers p_{clos} must in all cases be pushed onto stack W after the items that have been included in the closures have been removed.

We are now ready to study the step-by-step execution of the SASM code given in Fig. 10.9. The sequence of machine states (or stack configurations) produced by this code is shown in Fig. 10.10. Since this code runs forever, as we remember from the discussion of the equivalent high-level program in Sect. 10.1, we need to enforce termination by means of the count variable rr that we have chosen to initialize with the value 4, i.e., the machine performs four function calls and thereafter executes the remaining code as just described.

STEP	INSTR	S1	S2	S3	R	rr
0	ENTRY	$\mid W$	$\mid A$	$\mid T$	$\mid$	4
1	PUSH_W 2	2 $\mid W$	$\mid A$	$\mid T$	$\mid$	4
2	PUSH_W 1	1 2 $\mid W$	$\mid A$	$\mid T$	$\mid$	4
3	BRA_T p_f	1 2 $\mid T$	$\mid A$	$\mid W$	$\text{EX}^0\mid$	3
4	COPY_TW 0	1 2 $\mid T$	$\mid A$	1 $\mid W$	$\text{EX}^0\mid$	3
5	COPY_TW 1	1 2 $\mid T$	$\mid A$	2 1 $\mid W$	$\text{EX}^0\mid$	3
6	BRA_F p_g	1 2 $\mid T$	$\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
7	COPY_AW 0	1 2 $\mid T$	2 $\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
8	COPY_TW 0	1 2 $\mid T$	1 2 $\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
9	GT	1 2 $\mid T$	`true` $\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
10	JFALSE p_m	1 2 $\mid T$	$\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
11	COPY_AW 1	1 2 $\mid T$	1 $\mid W$	2 1 $\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
12	FREE_A 2	1 2 $\mid T$	1 $\mid W$	$\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
13	COPY_TW 2	1 2 $\mid T$	1 1 $\mid W$	$\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
14	PUSH_W 2	1 2 $\mid T$	1 1 1 $\mid W$	$\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
15	MINUS 2	1 2 $\mid T$	0 1 $\mid W$	$\mid A$	$\text{FT}^0 \ldots \mid \text{EX}^0\mid$	2
16	JTAIL p_g	1 2 $\mid T$	0 1 $\mid A$	$\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
17	COPY_AW 0	1 2 $\mid T$	0 1 $\mid A$	0 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
18	COPY_TW 0	1 2 $\mid T$	0 1 $\mid A$	1 0 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
19	GT	1 2 $\mid T$	0 1 $\mid A$	`false` $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
20	JFALSE p_m	1 2 $\mid T$	0 1 $\mid A$	$\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
21	COPY_AW 0	1 2 $\mid T$	0 1 $\mid A$	0 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
22	FREE_A 2	1 2 $\mid T$	$\mid A$	0 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
23	PUSH_W 1	1 2 $\mid T$	$\mid A$	1 0 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
24	ADD	1 2 $\mid T$	$\mid A$	1 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
25	COPY_TW 1	1 2 $\mid T$	$\mid A$	2 1 $\mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	1
26	BRA_T p_f	1 2 $\mid W$	$\mid A$	2 1 $\mid T$	$\text{RTF}^0 \mid \text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
27	COPY_TW 0	2 1 2 $\mid W$	$\mid A$	2 1 $\mid T$	$\text{RTF}^0 \mid \text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
28	COPY_TW 1	1 2 1 2 $\mid W$	$\mid A$	2 1 $\mid T$	$\text{RTF}^0 \mid \text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
29	BRA_F p_g	p_{clos} 1 2 $\mid W$	$\mid A$	2 1 $\mid T$	$\text{RTF}^0 \mid \text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
30	FREE_T 2	p_{clos} 1 2 $\mid W$	$\mid A$	$\mid T$	$\text{RTF}^0 \mid \text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
31	RTT	1 2 $\mid T$	$\mid A$	$p_{clos} \mid W$	$\text{FT}^1 \ldots \mid \text{EX}^0\mid$	0
32	RTF	1 2 $\mid T$	$\mid A$	$p_{clos} \mid W$	$\text{EX}^0\mid$	0
33	FREE_T 2	$\mid T$	$\mid A$	$p_{clos} \mid W$	$\text{EX}^0\mid$	0
34	RTT	$p_{clos} \mid W$	$\mid A$	$\mid T$	$\mid$	0
35	EXIT	$p_{clos} \mid W$	$\mid A$	$\mid T$	$\mid$	0

Fig. 10.10. Execution sequence of the SASM code of Fig. 10.9

The first column of the table in Fig. 10.9 simply enumerates state transition steps, and the second column lists the instructions that effect these transitions. The next three columns depict three memory sections $S1$, $S2$, $S3$ onto which the stacks W, A, T are mapped. The sixth column shows the return stack R, and the last column shows the value of the reduction counter rr. Each row of the table shows the stack contents and the permutation of the

stacks over the memory sections $S1$, $S2$, $S3$ immediately after execution of the instruction in the second column. All stacks grow to the left. In the case of this particular program the stacks contain just basic values, stack R shows the first instructions of the return continuations (abbreviated to EX for EXIT and FT for FREE_T), which are separated by '|' symbols. The superscripts attached to these instructions give the current values of the flag bits that keep track of the odd/even stack flips effected by JTAIL instructions.

The instruction sequence begins with the top-level piece of code that computes $(\sim f\ 1\ 2)$ in that it pushes both argument values onto stack W and then branches to the code of the function f (steps 1, 2 and 3). This code, in turn, computes $(g\ \sim 1\ \sim 0)$ by retrieving both arguments from stack T and then branching to the code of g (steps 4, 5 and 6).

The first action of g is to evaluate the predicate (GT $\sim$ 0 #0) of its conditional to true (steps 7, 8, 9) by moving to W an argument each from stacks A and T and then doing the $\geq$ comparison. Ignoring the branch in step 10, the computation then continues with the consequent code to compute the application $(g\ (-\ 1\ \sim 0)\ \#1)$ (steps 11 to 16), with JTAIL p_g effecting a jump back to the beginning of the code for g. We note that between steps 11 and 16 two entries of stack A that constitute the A-frame for g can be dropped since they are no longer needed.

The next instance of the code for g, beginning at step 17, now takes the branch at step 20 to continue with the code for $(\sim f\ \sim 1\ (+\ 1\ \#0))$ (steps 21 to 26).

The instructions BRA_T in steps 3 and 26 flip stacks T and W, the instructions BRA_F in step 6 and JTAIL in step 16 flip the stacks A and W, and the BRA_X instructions push new return continuations onto stack R with the flag bits set to 0.

Looking at the last column, we see that the count value rr is decremented by every branch and tail-jump instruction, and that BRA_T p_f at step 26 decrements this value to zero, i.e., the machine reaches the breakpoint at which regular instruction execution stops.

From here on, the machine turns all function calls into closures but executes all other instructions as usual. Thus the branch to p_f is taken and execution continues with the code for f. In steps 27 and 28, the arguments for another call of g are moved from stack T to W but the instruction BRA_F p_g in step 29 creates from the two topmost entries in W and T the closure

$$p_{clos} \rightsquigarrow (\sim (p_g\ 1\ 2)\ 2\ 1)\ ,$$

and pushes the pointer p_{clos} onto W instead. Next, two entries are popped off stack T (step 30), and the instruction RTT of step 31 restores from stack R the return continuation consisting of the single instruction RTF. As the flag of this instruction is set to 0, RTT effects a stack flip between T and W that also moves the pointer p_{clos} to the top of what has now become stack W. When RTF is executed next, the return continuation in R is found to be flagged out with 1 which, without a stack switch, returns control back to the

remaining code of the function f. This piece of code releases two more entries from stack T and then returns, after another stack flip between T and W, to the remainder of the top-line code, which is just the instruction EXIT (steps 33 to 35).

As we can see, the program terminates after four function calls with the closure to which the single entry p_{clos} in stack W points, with all other stacks empty, and with the stacks permuted over the memory sections $S1$, $S2$, $S3$ in the same way as at the beginning of code execution.

The postprocessor transforms the resulting closure into the high-level program

```
letrec
   g = λwz.if( gt 2 w ) then( g ( − 1 2 ) z )
                        else letrec f = ...in ( f 1 ( + 1 w ) )
in ( g 1 2 )
```

by undoing the tilde application and reconstructing, through the descriptor to which p_g is pointing, the nesting of `letrec`s (and also the variable names) as it defines a partially instantiated variant of the function g that has its nonlocal parameters u, v substituted by the values 2, 1, respectively.

10.5 Reducing to Full Normal Forms ∗

Both the LASM and the SASM are only weakly normalizing. Partial applications of abstractions, including as special cases unapplied abstractions, that pop to top level are considered irreducible and are turned into closures, the reason being that abstraction codes of both machines are static and therefore need to be supplied with full sets of arguments in order to execute correctly. These closures could simply be left as they are and unraveled by the postprocessor, based on the structural information and on the variables contained in the persistent graphs of the original programs (see Fig. 10.3), to produce as output equivalent high-level expressions.

However, with a little more effort we can do a lot better than that. All that it takes to fully normalize partial applications or unapplied abstractions is to η-extend them to full applications. This is in fact equivalent to what the $\lambda\sigma$-machine of Sect. 6.3 does when it applies the *beta*-rule to update substitutions that penetrate the scopes of abstractors, and it is also equivalent to how the head-order machines of Chaps. 7 and 8 handle partial applications.

We remember that in both cases the unapplied Λ-abstractors, in the course of η-extensions, end up in a leading *lambs* sequence that never engages in further β-reductions and therefore becomes part of the full normal form with which the computation hopefully terminates eventually. Since π–RED is intended to return as output high-level expressions in which all variables and function identifiers are restored as in the original program expressions, we

can take advantage of this property by using the η-extensions to reintroduce the original variables and λ-abstractors. All we need to do conceptually is to transform some partially applied abstraction that, as a Λ-expression, has the form[11]

$$(\underbrace{\Lambda \ldots \Lambda}_{n}.e_0\ e_1 \ldots e_k) \ \text{ with } \ k < n$$

into an $(n-k)$-ary abstraction

$$\lambda v_{k+1} \ldots v_n.((\underbrace{\Lambda \ldots \Lambda}_{n}.e_0\ e_1 \ldots e_k)\ v_{k+1} \ldots v_n) \ ,$$

in which the missing arguments are replaced by the bound variables themselves.[12] In π–RED these variables may be recovered from the persistent program structures set up initially in the heap. The body of this new abstraction can now be weakly normalized again by executing the code of the abstraction $\underbrace{\Lambda \ldots \Lambda}_{n}.\ e_0$. The new abstractor $\lambda v_{k+1} \ldots v_n$ (which is the equivalent of the leading *lambs* sequences produced by the head-order reducers of Chaps. 7 and 8 and by the $\lambda\sigma$-machine) must be kept in a separate structure, from which it may be retrieved when constructing the resulting expressions.

A partial application may be encountered by π–RED when executing a code fragment

$$\ldots ;\ \text{PUSH\_W}\ p_f;\ \text{AP}\ k;\ \ldots \ .$$

As illustrated in Fig. 10.3, p_f points to a function descriptor that in turn includes a link each to the function code and to the list of its λ-bound variables, which in the case of the above abstraction would be $< s\ n \mid v_1 \ldots v_n >$ (with s denoting the number of relatively free variables that have been λ-lifted). The instruction AP k inspects this descriptor for the value of n and, if found smaller than its own parameter k, removes the pointer p_f and k arguments $a_1 \ldots a_k$ of the application from stack W and creates in its place a closure of the form

$$p_{clos} \ \rightsquigarrow \ (^{\sim}\ (p_f\ a_1 \ldots a_k)\ b_1 \ldots b_s) \ ,$$

where $b_1 \ldots b_s$ denote the entries of the topmost T-frame. Whenever such a closure occurs at the top level, it is turned over for further processing to the η-extension mechanism included in Fig. 10.1. This mechanism provides another stack P that keeps track of nested η-extensions but it has also access to the machine stacks A, W, T, R. By interpretation of the components of such a closure, this mechanism

[11] Of course, in π–RED the abstraction $\underbrace{\Lambda \ldots \Lambda}_{n}.\ e_0$ is compiled to a piece of code.

[12] In the special case of an unapplied abstraction we have $k = 0$, i.e., η-extension trivially returns the original abstraction $\lambda v_1 \ldots \lambda v_n.e_0$ itself.

- creates in stack W a full argument frame by pushing first the variables $v_n, \ldots, v_{k+1}$ retrieved from the variable list held in the function's descriptor, followed by the arguments $a_k, \ldots, a_1$, both in the order from left to right;
- pushes the variables $v_n, \ldots, v_{k+1}$ in the same order also into stack P, from where the leading abstractors of the full normal form are constructed later on;
- creates in stack T a tilde frame by pushing the arguments $b_s \ldots b_1$ in that order;
- returns control back to the LASM or the SASM proper by executing (the equivalent of) a BRA_F p_f instruction to branch to and execute the function code in the environment defined by what, after the stack switch effected by this instruction, have become the topmost A- and T-frames.

Such η-extensions may have to be repeated several times until an expression is fully normalized.

The variables that in an η-extension step are pushed onto stack P must be popped again when returning, by means of the instruction RTF, from a partially applied function to construct the leading λ-abstractors that must precede the full normal form of the η-extended function application. The λ-abstraction thus obtained is then placed in the heap.

To remove the correct number of variables from stack P in each such step, the BRA_F instruction through which the function code is entered must push a separation symbol \$ onto stack P, which is removed by the complementary RTF instruction after all the variables found on top of it have been turned into abstractors.

There is one more thing that must be taken care of when η-extending partial applications as described. Reintroducing in this naive form the variables that have been extracted by the preprocessor may cause name clashes. They may be avoided by assigning to all variables restored in a particular η-extension step the same η-nesting index, beginning, say, with the index 1 for the first η-extension and incrementing. Thus, when repeatedly going through the cycle of η-extension and code execution, the machine never needs to engage in full-fledged β-reductions to maintain correct bindings among occurrences of identically named variables that are bound in different contexts.

To illustrate how all this works, we consider as an example the nonterminating program expression

$$\texttt{letrec}\ f = \lambda uv.(+\ (f\ u)\ v)\ \texttt{in}\ (f\ v)$$

that includes two partial applications of the binary function f. When β-reducing the body expression $(f\ v)$ of the `letrec` following the classical definition given in Sect. 4.2, we obtain

$$\lambda v^1.(+\ (\texttt{letrec}\ f = \lambda uv.(+\ (f\ u)\ v)\ \texttt{in}\ (f\ v)\ v^1)\ .$$

Here we have α-converted to v^1 the λ-bound variable v that otherwise would get caught in a naming conflict with the free occurrence of v in operand position. Repeating this step one more time gives

$$\lambda v^1.(+\ \lambda v^2.(\texttt{letrec}\ f = \lambda uv.(+\ (f\ u)\ v)\ \texttt{in}\ (f\ v)\ v^2)\ v^1)\ .$$

The SASM that we will consider here since it is a little simpler than the LASM and serves the intended purpose well enough, has this λ-expression compiled to the code

$p_e \rightsquigarrow$ PUSH_W p_v; PUSH_W p_f; AP 1; EXIT;
$p_f \rightsquigarrow$ COPY_AW 1; COPY_AW 0; FREE_A 2; PUSH_W p_f; AP 1; ADD; RTF; .

To perform the same number of β-reductions as above, the count value rr needs to be initialized to $rr = 2$, which allows for two function calls before it is forced to stop.

Figure 10.11 shows the sequence of stack configurations produced by this code, including the two η-extension steps that become necessary. The layout of the figure is essentially the same as that of Fig. 10.10, except that stack T has been dropped since it does not engage in this particular computation.

The first piece of code executed in steps 1 to 4 creates a closure for $(f\ v)$, and the pointer p_{clos} to it becomes the sole entry in W, i.e., the partial application is at the top level. With nothing else left to do, the machine calls the η-extension mechanism that pushes onto W the η-extended variable v^1 followed by v to prepare f for a full application. It also pushes v^1 onto the η-extension stack P. The first part of the η-extension is completed in step 5 by executing a branch instruction that returns control back to the SASM to execute the code of the function f. The branch also effects a stack switch between W and A, stores a return continuation in R, and decrements the reduction counter rr by one.

The code of f executed in steps 6 to 10 creates a second closure for another instance of the partial application $(f\ v)$ that again is η-extended by pushing onto stack W the η-indexed variable v^2 followed by v, by pushing v^2 also onto P, and then calling the code for f again in step 11.

Since this η-extension step brings the count value rr down to zero, the code that is left is trivially executed by turning without further action all remaining redices into closures. The first such redex is another instance of $(f\ v)$ that in step 16 is wrapped up in a closure referenced by the pointer p^1_{clos}, immediately followed by the creation in step 17 of a closure for the expression $(+\ p^1_{clos}\ v^2)$, to which p^2_{clos} points.

The RTF instruction of step 18, after having flipped the stacks W and A and restored a return continuation from stack R, calls the η-extension mechanism again to complete the η-extension of step 11. It does so by using the indexed variable v^2 popped off stack P to prepend the abstractor λv^2 to the expression pointed to by p^2_{clos}.

STEP	INSTR	S1	S2	R	P rr
1	ENTRY	$\mid W$	$\mid A$	$\mid$	$\mid 2$
2	PUSH_W v	$v \mid W$	$\mid A$	$\mid$	$\mid 2$
3	PUSH_W p_f	$p_f\ v \mid W$	$\mid A$	$\mid$	$\mid 2$
4	AP 1	$p_{clos} \mid W$	$\mid A$	$\mid$	$\mid 2$
	η-extending the closure $p_{clos} \rightsquigarrow (\ p_f\ v\ )$				
	new stack configuration				
		$v\ v^1 \mid W$	$\mid A$	$\mid$	$v^1\ \$ \mid 2$
	branching to the function code				
5	(BRA_F p_f)	$v\ v^1 \mid A$	$\mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
6	COPY_AW 1	$v\ v^1 \mid A$	$v^1 \mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
7	COPY_AW 0	$v\ v^1 \mid A$	$v\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
8	FREE_A 2	$\mid A$	$v\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
9	PUSH_W p_f	$\mid A$	$p_f\ v\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
10	AP 1	$\mid A$	$p_{clos}\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^1\ \$ \mid 1$
	η-extending the closure $p_{clos} \rightsquigarrow (\ p_f\ v\ )$				
	new stack configuration				
		$\mid A$	$v\ v^2\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 1$
	branching to the function code				
11	(BRA_F p_f)	$\mid W$	$v\ v^2\ v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
12	COPY_AW 1	$v^2 \mid W$	$v\ v^2\ v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
13	COPY_AW 0	$v\ v^2 \mid W$	$v\ v^2\ v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
14	FREE_A 2	$v\ v^2 \mid W$	$v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
15	PUSH_W p_f	$p_f\ v\ v^2 \mid W$	$v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
16	AP 1	$p^1_{clos}\ v^2 \mid W$	$v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
17	ADD	$p^2_{clos} \mid W$	$v^1 \mid A$	$\text{ADD}^0 \ldots \text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
18	RTF	$\mid A$	$p^2_{clos}\ v^1 \mid W$	$\text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
	completing the η-extension of step 11				
	$p^2_{clos} \rightsquigarrow (\ + p^1_{clos}\ v^2\ ) \Longrightarrow_\eta p^2_{clos} \rightsquigarrow \lambda v^2.(\ + p^1_{clos}\ v^2\ )$				
19	ADD	$\mid A$	$p^3_{clos} \mid W$	$\text{EX}^0 \mid$	$v^2\ \$\ v^1\ \$ \mid 0$
20	RTF	$p^3_{clos} \mid W$	$\mid A$	$\mid$	$v^1\ \$ \mid 0$
	completing the η-extension of step 5				
	$p^3_{clos} \rightsquigarrow (\ + p^2_{clos}\ v^1\ ) \Longrightarrow_\eta p^3_{clos} \rightsquigarrow \lambda v^1.(\ + p^2_{clos}\ v^1\ )$				
21	EXIT	$p^3_{clos} \mid W$	$\mid A$	$\mid$	$\mid 0$

Fig. 10.11. Executing SASM code that requires η-extensions

Steps 19 and 20 create another closure for $(+\ p^2_{clos}\ v^1)$ and prepend the abstractor λv^1 to it; the pointer p^3_{clos} to it remains the sole entry in W when executing in step 21 the EXIT instruction.

The full picture of what we are getting develops when we follow the chain of pointers, beginning with p^3_{clos}. It gives

$$p^3_{clos} \rightsquigarrow \lambda v^1.(+\ p^2_{clos}\ v^1) \mid p^2_{clos} \ \rightsquigarrow\ \lambda v^2.(+\ p^1_{clos}\ v^2) \mid p^1_{clos} \ \rightsquigarrow\ (p_f\ v)$$

or, when these pointers are dereferenced and substituted by the expressions hidden behind them, the λ-expression

$$\lambda v^1.(+\ \lambda v^2.(+\ (p_f\ v)\ v^2)\ v^1) \ ,$$

which is what the postprocessor constructs. In a last step, the postprocessor dereferences the pointer p_f to replace the innermost application with the complete `letrec` expression, yielding, as expected,

$$\lambda v^1.(+\ \lambda v^2.(\texttt{letrec}\ f = \lambda uv.(+\ (f\ u)\ v)\ \texttt{in}\ (f\ v)\ v^2)\ v^1)$$

as the intermediate expression after two (partial) function applications.

Converting the bound variables v^1 and v^2 into v by means of the α-conversion function $\lambda u.\lambda v.(u\ v)$, and using protection keys as introduced in Sect. 4.3 to distinguish different bindings of equally named variables, would yield

$$\lambda v.(+\ \lambda v.(\texttt{letrec}\ f = \lambda uv.(+\ (f\ u)\ v)\ \texttt{in}\ (f\ //v)\ v)\ v) \ ,$$

We note that the free variable occurrence v in the original expression preserves its binding status in that it is now protected against the two binders λv whose scope it has penetrated.

10.6 Summary

This chapter describes two variants of a fully normalizing reduction system π–RED, of which one realizes a lazy, the other a strict semantics. It appears to the user as a system that performs high-level program transformations governed by the reduction rules of a full-fledged applied λ-calculus. These transformations may, under interactive control, be carried out step by step, and intermediate program expressions may be displayed at the user interface.

Internally, the system executes compiled code composed of fairly conventional instructions that operate primarily on stacks. The underlying abstract machines essentially mimic the $\lambda\sigma$-calculus machine introduced in Sect. 6.3 insofar as weakly normalizing code execution phases may be interspersed with η-extensions equivalent to applications of the *beta*-rule that moves substitutions over abstractors. These η-extensions are to maintain correct variable bindings whenever argument expressions containing (relatively) free variables must be substituted under abstractors and normalization must continue in abstraction bodies.

The basic ideas of compiling high-level AL-like programs to code and of code execution are similar to those of the *G*-machine. Differences are primarily

due to another, less rigorous concept of closing λ-expressions. Rather than converting individual abstractions into supercombinators, π–RED employs a preprocessor to lift relatively free variables out of the larger contexts of `letrec` expressions and distinguishes them by different tags from the variables that are locally bound. These tagged variables are subsequently converted into different sets of binding indices that define fixed offsets in separate runtime structures.

Since π–RED supports stepwise program transformations, compilation to executable code must preserve the structure of the original program, specifically the nesting of function definitions, in order to be able to decompile intermediate states of code execution into high-level expressions that correspond to instances of β-reductions completed. To this end, the compiler generates a persistent graph that typically features alternatingly `letrec` nodes and abstraction nodes.

Both abstract machines use the same runtime structures of four cooperating stacks, but differ with regard to the use of these stacks, which also affects the compilation to abstract machine code.

Laziness requires that argument expressions be wrapped up in suspensions to postpone their evaluation until a demand for their normalization arises later on. Since several such suspensions may share the same environments, and the suspensions generally survive the lifetimes of the function calls that created them, the entire environment structure that must be kept alive during a program run typically looks like a cactus structure of frames, as in Fig. 7.7.

Things are decidedly simpler in the strict machine since the arguments of a function application are evaluated right away in the environment of the calling context, which renders suspensions superfluous. The environment frames may therefore be placed directly in the stacks and released as soon as control returns from the called to the calling context, i.e., creating and releasing frames follows a last-in-first-out (LIFO) order.

Compilation to executable code is slightly more complex in the lazy machine as it must generate instructions that create suspensions and subsequently force their evaluation; there are no such instructions in the equivalent code of the strict machine.

Both machines return graphs as intermediate or terminal states of computation. These graphs are converted into high-level output by a postprocessor that, with the help of the persistent graph structure generated by the compiler from the expression prepared by the preprocessor, reconstructs the original `letrec` nestings and function definitions that are still in use.

A complete program execution cycle thus includes preprocessing the input expression, compiling the preprocessed expression to code, typically going repeatedly through the subcycle of executing code and doing η-extensions to construct the graph of an intermediate or a fully normalized expression, and finally postprocessing this graph to return it as high-level output.

It is fairly straightforward to have any weakly normalizing machine participate in an execution cycle that accepts high-level program expressions as

input and returns partially reduced or fully normalized expressions in high-level notation as output.

Things remain nearly the same when the π–RED machines, specifically the lazy variant, are replaced by the G-machine. Differences primarily concern the preprocessor. It must convert individual λ-abstractions into supercombinators rather than closing entire `letrec` contexts, and when compiling the supercombinators to code the nestings of the original function definitions must somehow be preserved for possible reconstruction of high-level output, rather than lifting them all to the top level, as the original G-compiler does. G-code execution must be complemented by η-extensions exactly as in π–RED whenever top-level partial applications or unapplied supercombinators are encountered, and the postprocessor must reconvert whatever graphs are produced into high-level expressions. Supporting stepwise code execution may also preclude certain G-code optimizations since a one-to-one correspondence must be preserved between pieces of code and β-reductions, as the intermediate or final states of code execution must be decompiled into high-level expressions that correspond to β-reductions performed.

The B-machine of Chap. 8 conceptually looks like the better choice for this purpose. Since it is fully normalizing itself, a preprocessor would have to convert λ-bound variables of an AL-like high-level program expression into binding indices, but there would be no need to close abstractions. The compiler, in the course of translating function definitions into B-code, again would have to preserve the nestings of function definitions and the names of λ-bound variables. Code execution must not be interspersed with explicit η-extensions since this is implicitly taken care of by corrective actions on the binding indices whenever the runtime environment is accessed. A postprocessor would have to convert intermediate or final states of code execution into high-level output, which would have to include the reintroduction of λ-bound variables (which in π–RED is done as an integral part of η-extensions).

References

The description of the SASM was published in a paper by Gaertner and the present author [GK96]. An earlier version of this machine that directly interprets graph representations of λ-terms rather than executing code was published in [SBK92]. This paper also includes a more detailed description of pre- and postprocessing and of closing λ-terms. A description of the LASM is not published elsewhere. The basic concepts of π–RED, particularly support for a full-fledged λ-calculus and the idea of step-wise reductions, derive from Berkling's string reduction machine [Ber75].

The description of a parallel version of π–RED may be found in [HaMi99].

Work on fully normalizing abstract λ-calculus machines has also been published by Crègut [Cre90] and by Grégoire and Leroy [GrLe02].

The benefits of stepwise execution and inspection of intermediate program expressions have also been advocated by Goldson [Gol94]. More recent versions of LISP and SCHEME implementations support a similar form of stepwise execution, largely for debugging purposes, as well [Sun90, All92, PLTS96].

11

Pattern Matching

Pattern matching is a powerful mechanism that, roughly speaking, extracts (sub)structures from given structural contexts and substitutes them for placeholders in other (structural) contexts. It is typically used in functional languages to define as ordered sets of alternative function equations restructuring operations on lists. In a more general setting, pattern matching may also be used in the area of rule-based transformations of constructor expressions that in fact introduce a meta-language level on which these transformations can safely be carried out without corrupting the Church–Rosser property and referential transparency of the underlying functional language. This approach may be effectively employed to quickly prototype, as sets of pattern matching functions specified on this meta-language level, compilers or language interpreters such as the abstract machines discussed in this text, or to implement term rewrite systems and, as we will see in Appendix B, theorem provers.

In this chapter, we will briefly discuss how AL may be extended by pattern matching on n-ary lists, how this concept can be generalized to specify other languages and language interpreters as rule-based transformations of constructor expressions, and how the lazy version LASM of the π–RED machines described in the preceding chapter may be used to implement it. This machine is of particular interest in that it is fully normalizing on the one hand and supports a stepwise execution mode on the other hand, with the amenities of modifying intermediate expressions and shifting the scope of reductions to selected subexpressions – properties that play an important role in theorem proving.

11.1 Pattern Matching in AL

In its simplest form, pattern matching refers to the application of what may be called a pattern abstraction to an argument. Pattern abstractions differ from ordinary λ-abstractions in that they have the abstractors $\lambda u_1 \ldots u_n$ replaced

by patterns specified as list structures whose components are variables, numerical values, character strings or recursively again patterns. Just like λ-bound variables, the pattern variables are said to be bound in the pattern abstraction bodies.

Pattern matching compares the structure of such a pattern with the structure of the argument to which it is applied. It is said to succeed if for every (sub)pattern there is a corresponding (sub)structure in the argument, every pattern variable can be associated with a (sub)structure and every numerical and string value equals a value and a string in the corresponding syntactical position of the argument. Otherwise it is said to fail.

If the match succeeds, every free occurrence of a pattern-bound variable in the body of the pattern abstraction is substituted by the matching (sub)structure of the argument, and evaluation continues in the body thus instantiated.

If the match fails, one can try alternative pattern abstractions and, if all of them fail, turn to (the evaluation of) yet another expression that serves as some kind of an escape hatch.

The patterns may be supplemented by guard expressions that specify conditions which, in addition to succeeding matches, must be satisfied by the (sub)structures substituted for pattern-bound variables.

Pattern matching on n-ary lists raises the general problem that patterns may have to be specified for (sub)structures of varying lengths that may also occur in varying syntactical positions. To locate and match against such (sub)structures, the patterns may have to include wild cards that bind or simply skip over varying numbers of list components.

A typical example would be a test for the palindrome property of a list of numbers, say $< 4\ 5\ 3\ 7\ 3\ 5\ 4 >$. To tackle this problem, it would be convenient to be able to specify a pattern of the form $< u\ \texttt{as}[w]\ v >$, together with a guard term $(\texttt{eq}\ u\ v)$. When applied to the above list, this pattern would have to bind the first element 4 to the pattern variable u, the last element 4 to the pattern variable v, and the sublist $< 5\ 3\ 7\ 3\ 5 >$ of elements that are in between to the variable w, denoted as a wild card by the key word `as`. As the guard term would evaluate to `true`, the pattern match would succeed and could be recursively applied to the list bound to w until it either ends up with an empty or one-element list signifying the palindrome property, or fails with an unsatisfiable guard.

The palindrome problem is just a simple example of using wild cards between two pattern variables. The more general case would have wild cards anywhere in a pattern or in subpatterns. It takes little imagination to realize that translating such patterns into abstract machine code would require a set of suitable instructions and also a rather elaborate compilation scheme. To keep things simple and convey just the basic ideas of implementing pattern matches on the LASM, we therefore allow wild cards to be used only in the last (rightmost) syntactical position of a pattern. Such patterns are good enough

to extract one or several leading elements from an n-ary list and to bind the rest to a wild-card variable.

With these things in mind, we can define an AL-compatible syntax for pattern matching as follows: [1]

$$
\begin{aligned}
e =_s\ & \texttt{case}\ pattern_1 \parallel guard_1 \rightarrow e_1 \\
& \quad \ldots \\
& \quad pattern_n \parallel guard_n \rightarrow e_n \\
& \texttt{otherwise}\ e_0 \\
pattern =_s\ & val \mid string \mid var \mid < pattern\ \{pattern\}^*\ \{\texttt{as}[\ var\ ]\} > \mid \\
& label[\ pattern\ \{pattern\}^*\ \{\texttt{as}[\ var\ ]\}\ ]\ .
\end{aligned}
$$

The `case` construct in fact realizes a complex unary function composed of an ordered set of n pattern abstractions, including guards, followed by an `otherwise` expression that serves as the aforementioned escape hatch. When this construct is applied to an argument, the pattern abstractions are tried in the order from top to bottom, and the first pattern that matches and satisfies the guard[2] has the entire `case` application reduced to the associated body expression in which all free occurrences of the pattern variables are substituted by the matching components of the argument. If none of the patterns matches, the application reduces to the (value of) the `otherwise` expression e_0.

As said before, a pattern is either a numerical value *val*, a variable *var*, a character string *string* embedded in quotes, or a list of these items that may recursively contain patterns as elements. The variables that occur free in the body expression or in the guard of a pattern abstraction may be either locally bound by the pattern, nonlocally in a surrounding context, or free in the entire program expression. The patterns are assumed to be linear, meaning that all pattern-bound variables of a pattern abstraction must be unique.

A wild card `as`[*var*] that may occur as the last (rightmost) component of a pattern collects in a new list the elements of the argument list that occur to the right of the elements that match preceding subpatterns.

Labeled patterns *label*[*pattern_1* ...] are just syntactic sugar for list patterns of the form < '*label*' *pattern_1* ... > that have labels in the form of strings in their first components. These labels may be considered customized constructors that may be effectively employed to construct the terms of some meta-language that sits on top of AL proper and can be interpreted by sets of pattern abstractions specified in this notation.

11.2 Programming with Pattern Matches

A few examples of increasing complexity may illustrate how pattern matching is supposed to work.

[1] As usual, {*item*}* denotes zero or more occurrences of *item*, and {*item*} denotes zero or one occurrences of *item*.

[2] A guard may be any expression that evaluates to either `true` or `false`.

The `case`-function

- `case`
  $< x\ `aa' < v\ u >\ w >\ \|\ (\texttt{eq}\ x\ v) \rightarrow < x\ w >$
  `otherwise` $`no\ match'$

matches the argument $< 4\ `aa'\ < 4\ 7 >\ 3 >$, returning the list $< 4\ 3 >$, but fails on the argument $< `aa'\ < 4\ 7 >\ 3 >$ due to mismatching arities, on $< 4\ `bb'\ < 4\ 7 >\ 3 >$ due to a mismatch between the strings $`aa'$ and $`bb'$, or on $< 2\ `aa'\ < 4\ 7 >\ 3 >$ due to an unsatisfiable guard, returning in each of these cases the string value $`no\ match'$.

The `case` function

- `case`
  $< x\ < u\ v >\ \texttt{as}[w] >\ \|\ \texttt{true} \rightarrow < w\ < u\ x >\ v >$
  `otherwise` $`no\ match'$

matches the argument $< y\ < `aa'\ `bb' >\ 2\ 3\ 4 >$, returning as the result the list $<< 2\ 3\ 4 > < `aa'\ y >\ `bb' >$, but fails, for instance, on the argument $< 2\ < 5\ 6\ 7 > `cc' >$ due to mismatching arities between the subpattern $< u\ x >$ and the sublist $< 5\ 6\ 7 >$.

A more involved example that uses wild cards is a recursive function that figures out whether or not two lists are identical with regard to the numbers of elements and the elements themselves:

- `letrec`
  $compare = \texttt{lambda}\ u\ v\ \texttt{in}$
  (`case`
  $<<><<>>\ \|\ \texttt{true} \rightarrow `identity'$
  $<<>< w\ \texttt{as}[z] >>\ \|\ \texttt{true} \rightarrow `mismatching\ arities'$
  $<< w\ \texttt{as}[z] ><>>\ \|\ \texttt{true} \rightarrow `mismatching\ arities'$
  $<< u\_h\ \texttt{as}[u\_t] >< v\_h\ \texttt{as}[v\_t] >>\ \|\ (\texttt{eq}\ u\_h\ v\_h) \rightarrow (compare\ u\_t\ v\_t)$
  `otherwise` $`mismatching\ elements'$
  $< u\ v >$)
  `in` $(compare\ < 4\ 7\ 3\ \ldots >< 4\ 7\ 3\ \ldots >)$.

The function *compare* substitutes for the parameters u and v a first and a second argument that are both assumed to be lists of numbers. These lists become the components of a binary list (second last line) to which the `case` function in the body of *compare* is applied. This function includes four pattern abstractions, of which the first checks whether both sublists are empty, signifying identity of the two argument lists, the second and the third check whether one of the sublists is empty while the other contains at least one element, signifying a mismatch with regard to the lengths of both argument lists, and the fourth one applies *compare* recursively to the argument lists minus their first elements if these are identical.

The use of labeled patterns may be best demonstrated by means of an AL specification of the SE(M)CD machine of Chap. 5 that implements the pure λ-calculus. Its three syntactical figures may be represented as customized constructor terms

$$cct \ =_s \ var[\ `v'\ ] \mid lam[\ `v'\ cct\ ] \mid apa[\ cct\_a\ cct\_f\ ] \mid apn[\ cct\_f\ cct\_a\ ]$$

of a meta-language on top of AL, where *var*, *lam*, *apa*, *apn* are constructor labels for variables, abstractions, applicative and normal order applications, respectively. Specific variables, as may be noted, must be quoted in this language in order to distinguish them from the AL variables proper that are used in the patterns themselves.[3]

Thus, a λ-expression

$$@\ @\ \lambda u\ \lambda v\ \overline{@}\ v\ u\ \ \lambda u\ u\ \ w$$

as it can be evaluated by the SE(M)CD machine of Chap. 5 must be converted into

$$apn[\ apn\ [\ lam[\ `u'\ lam[\ `v'\ apa[\ var[`v']\ var[`u']\ ]\ ]\ ] \\ lam[\ `u'\ var[`u']\ ]\ ]var[\ `w'\ ]\ ]$$

for interpretation by AL-style pattern matching.

Another constructor term *clos*[*env cct*] is needed to represent closures, and we will also use the terms *apa*[*num*] and *apn*[*num*] to represent applicators with associated arity indices *num* as they occur isolated from their component terms in stack M of the SE(M)CD machine.

With this in mind, we are now ready to transliterate, more or less one-to-one, the state transition rules of the SE(M)CD machine as they are given in Fig. 5.5 into a set of pattern abstractions wrapped up in a `case` construct.

Just a few representative pattern abstractions are given in Fig. 11.1. From top to bottom, they implement rules (1), (2), (5), (9) and (11) of Fig. 5.5. They all have the general form

$$< S\ E\ M\ C\ D\ > \ \| \ guard \ \rightarrow < S'\ E'\ M'\ C'\ D' > \ ,$$

with S, E, M, C, D denoting the pattern variables that fill in for the sublists that represent the respective stacks. More specific subpatterns such as $< apa[\ i\ ]\ M >$ or $< clos[E\ lam[v\ e]]\ S >$ expose *apa*[*i*] or *clos*[*E lam*[*v e*]] as the topmost items on stacks M and S, respectively.

Of course, the full SE(M)CD interpreter must have this `case` construct embedded in a recursive function that applies it to an initial state

$$< <> <> <> < \ cct\_term \ <> > <> >$$

[3] Internally these terms are transformed into lists whose first components have the labels quoted as well.

```
case
   < S E M  <> < Ee Cc Dd >>  ‖ true  → < S Ee M Cc Dd >

   << clos[Ee lam[v e]]  < ea S >>  E  < apa[0] M >  C D >  ‖ true
                  → < S  << v ea >  Ee >  M  < e  <>>< E C D >>
                  ...
   < S E  < apa[i] M > < var[v] C >  D >  ‖ (gt 0 i)
                  →  (< (lookup v E) S >  E  < apa[(− 1 i)] M >  C D >
                  ...
   < S E  < apa[i] M > < lam[v e] C >  D >  ‖ (gt 0 i)
                  → << clos[E lam[v e]] S >  E  < apa[(− 1 i)] M >  C D >
                  ...
   < S E M  < apa[e_a e_f] C >  D >
                  → < S E  < apa[2] M > < e_a  < e_f C >>  D >
                  ...
otherwise ...
```

Fig. 11.1. A pattern matching implementation of the SE(M)CD machine's state transition rules

and repeatedly to intermediate states until it terminates with

$$<<\ cct\_wnf\ <>><><><><><>>\ ,$$

which has the weak normal form *cct_wnf* of the initial term *cct_term* in the sublist S, and all other sublists empty.

11.3 Preprocessing Pattern Matches

The preprocessing of `case` constructs (or functions for that matter) before compilation to LASM machine code can in large part be adopted from what has been said in Sect. 10.1 about preprocessing ordinary AL expressions.

As a first preprocessing step, `case`s are turned into tilde applications by lifting all relatively free variables out in front, in the same way as they are lifted out of `letrec`s.

For instance, a `case` construct

```
case
     < u v >  ‖ (eq u z)  → << u w > < v z >>
     < u  < v w >>  ‖ (gt v z)  → < w  < u w >  z >
otherwise < w z > ,
```

of two pattern abstractions in which the variables w and z occur free[4] thus transforms into the tilde application

[4] Note that w also occurs bound in the second pattern abstraction.

$$
\begin{array}{l}
(\sim \tilde{\lambda} wz.\texttt{case} \\
\qquad\qquad < u\ v > \ \| \ (\texttt{eq}\ u\ z) \ \rightarrow << u\ w > < v\ z >> \\
\qquad\qquad < u\ < v\ w >> \ \| \ (\texttt{gt}\ v\ z) \ \rightarrow < w\ < u\ w >\ z > \\
\qquad\ \texttt{otherwise} < w\ z > \\
w\ z\ ) \ .
\end{array}
$$

Converting the $\tilde{\lambda}$-bound variables into T-indices follows the same rules as for `letrec` expressions.

However, the story is a little different for the variables bound by pattern abstractions. To relate the A-indices by which we wish to replace occurrences of pattern-bound variables in the abstraction bodies in a meaningful way to offsets into A-frames, we need to be a little more specific about how pattern matching is supposed to work on a step-by-step basis suitable for implementation in the LASM.

The basic idea here is to walk side by side through both the pattern and the argument, most conveniently under the control of a preorder traversal, and to determine for each symbol encountered in the pattern whether or not there is a matching symbol or substructure in the argument. This entails also the order in which pattern variables may be instantiated with (pointers to) matching argument (sub)structures and in which these instantiations may be temporarily pushed onto the workspace stack W. From there, they may either be dumped again if the matches fail or be moved into new A-frames allocated in the heap if the matches are successful.

To have the A-indices that replace occurrences of pattern-bound variables define the same offsets relative to the same A-frame bases as in the case of ordinary λ-abstractions, the following conversion rule must be used.

Let $u_1, \ldots,\ u_i, \ldots,\ u_n$ denote the sequence of pattern variables encountered when traversing a pattern in preorder, then free occurrences of $u_i \mid i \in \{1, \ldots,\ n\}$ in both the guard and the body of the respective pattern abstraction must be replaced with indices $\#(n - i)$, i.e., the order of indices must be the reverse of that for λ-bound variables (see Sect. 10.1).

The variables in the patterns themselves must simply be replaced by nameless binding symbols $\Box$ that serve the same purpose of just denoting syntactical positions as the anonymous abstractors Λ and $\tilde{\Lambda}$ of ordinary and tilde abstractions.

Following this conversion rule, we obtain for the example above the fully preprocessed tilde application of the `case` function as follows:

$$
\begin{array}{l}
(\sim \tilde{\Lambda}\tilde{\Lambda}.\texttt{case} \\
\qquad\quad < \Box\ \Box > \ \| \ (\texttt{eq}\ \#1\ \sim 1) \ \rightarrow << \#1\ \sim 0 > < \#0\ \sim 1 >> \\
\qquad\quad < \Box\ < \Box\ \Box >> \ \| \ (\texttt{gt}\ \#1\ \sim 1) \ \rightarrow < \#0\ < \#2\ \#0 > \sim 1 > \\
\qquad\ \texttt{otherwise} <\sim 0\ \sim 1 > \\
\qquad\ \#w\ \#z\ )
\end{array}
$$

(the symbols $\#w$ and $\#z$ in the argument positions of the tilde application (last line) are to indicate that the abstracted variables w and z may be bound in some larger context in which they are replaced with A- or T-binding indices as well).

As a final remark, it should be noted that the pattern abstractions are thus closed with respect to pattern variables (or A-indices), and that there is of course no notion of partially applying pattern abstractions since they are in fact unary functions in which `all` pattern variables must be instantiated with argument components for the matches to succeed.

11.4 The Pattern Matching Machinery

We will now investigate how pattern matching can be supported by the the lazy variant LASM of the π–RED machines described in the preceding chapter. It turns out that, other than assigning a more engaged role to stack R and requiring an additional set of dedicated instructions, pattern matching can be implemented on this machine in essentially the same way as ordinary function calls.

When reducing the tilde application of a closed `case` function, a T-frame is set up in the heap for instantiations of what were originally its (relatively) free variables, and the pointer to it is pushed onto stack T. While a particular pattern is being matched against a given argument, instantiations of the pattern variables, i.e., (pointers to) the matching argument components, build up successively on stack W. In the case of a successful match these entries are removed from stack W and placed in an A-frame created in the heap, and a pointer to this frame is pushed onto stack A, thus completing the environment in which both the guard and the body of the pattern abstraction must be evaluated.

Mismatches may occur with respect to the arities of the argument (sub)-structures, numerical values or strings may not be the same, or guards cannot be satisfied. In each of these cases, the match must be aborted, instantiations of the pattern variables that have thus far piled up in stack W or have already led to the creation of complete A-frames must be removed again, and code execution must continue with the next pattern abstraction in sequence.[5]

To support such pattern matches, the machine needs to keep track of nesting levels, index positions and numbers of bindings performed while it traverses an argument structure. This can be accomplished using stack R as follows:

- When entering an argument structure (or substructure) held as a graph in the heap H, the pointer to it is pushed onto stack R, followed by an index tuple $(i,\ l)$ that is initialized with $i = 0,\ l = 0$ and pushed on top.

[5] A pattern variable always matches if there is a corresponding syntactical position in the argument, and so do wild cards that also match empty structures.

- While moving along the components of a (sub)structure, the index i of the tuple that is on top of R keeps track of the position at which the matching process has arrived, and l counts the number of bindings completed up to and including position i.
- When a match on a (sub)structure has been successfully completed, the topmost index tuple and the pointer that is underneath are taken off stack R, and the component l of this tuple is added to that of the index tuple that has now popped to the top, i.e., the bindings on lower pattern levels accumulate on the higher levels.
- Failure to match a (sub)pattern at some nesting level causes the topmost index tuple $(i,\ l)$ and the pointer underneath to be popped off stack R, and l entries to be removed from stack W. The failure is propagated to the next higher nesting level, where the same operations are repeated until all entries in R pertaining to the particular pattern match have gone.

Otherwise, stack R is used as usual to store return addresses of ordinary function calls (applications of λ-abstractions) and of `case` applications. In the latter case, a return address is pushed before entry into the first pattern abstraction and popped after completing the evaluation either of the body expression of the matching pattern or, if no match succeeds, of the `otherwise` expression.

The instructions that are necessary to implement pattern matching as compiled code derive from the internal representation of (nested) lists, the preorder traversal scheme used by the matching process, and the steps that need to be taken to deal with succeeding or failing matches.

To simplify things a little, it suffices to assume that a nested list, say $< a_{11}\ a_{12}\ < a_{21}\ < a_{31}\ a_{32}\ a_{33} >\ a_{23} >\ a_{14} >$, whose components a_{ij} are atomic, is internally represented as a graph as depicted in Fig. 11.2. The (sub)lists of a particular nesting level are represented by pointers at the next higher level, and their arities are directly accessible as superscripts attached to the list constructors, i.e., list descriptors preceding the (sub)lists are ignored.

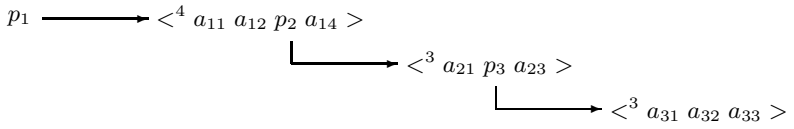


Fig. 11.2. Graph representation of a nested list structure

The following is the set of instructions used to implement pattern matches:

INIT_R is the first instruction of a pattern abstraction that initializes stack R. It pushes some dummy pointer $\diamond$, followed by an index tuple (0, 0) and by a copy of the argument (pointer) that is assumed to be on top of stack W, to set R up for the traversal of the argument.

IS_LIST n tests whether or not the item on top of stack R is a pointer to a list of arity n, pushing a `succ` or `fail` entry onto stack W, respectively, or, if the test cannot be decided since the entry on R is a pointer to a suspension, forcing its evaluation.

IS_STRING *string* tests whether the topmost entry of stack R is a character string *string* (or a numerical value given as a string of digits), taking the same actions as IS_LIST n if the entry indeed equals *string*, is a pointer to a closure, or is something else.

JPFAIL *pp* follows an IS_LIST or IS_STRING instruction. After having inspected the top item of stack W (which is assumed to be either `succ` or `fail`), code execution continues either at the next instruction in sequence or at the code referenced by the pointer *pp*. Before branching to *pp*, the instruction also undoes all the bindings that have piled up in W. It does so by popping entries $(i,\ l)$ and the pointers underneath off stack R, and in each such step it also pops l entries off stack W, down to and including the dummy pointer set up by INIT_R.

NEXT moves control to the next item of a (sub)list, copies it onto the top of stack R and increments the traversal position index i of the topmost tuple $(i,\ l)$ in R.

BIND follows NEXT if the topmost entry on stack R is to be substituted for a pattern variable (by moving it to the top of stack W), and increments the binding index l of the topmost $(i,\ l)$ tuple in R.

DOWN follows an IS_LIST; JPFAIL;... instruction sequence to descend, in the case of a successful match, to the next lower pattern level by pushing a new tuple $(0,\ 0)$ onto R on top of the pointer to the corresponding argument (sub)structure that is already in R.

UP is complementary to DOWN, returning control, after successfully completing a match on some sublist, to the next higher pattern level. It does so by taking the actual $(i,\ l)$ entry and the pointer underneath off stack R, and by changing the tuple $(ii,\ ll)$ that then pops to the top to $(ii+1,\ ll+l)$, thus advancing the traversal position at the next higher pattern level and updating the number of bindings accumulated thus far.

Another four instructions are required to bring about (and undo) the environments in which guard terms need to be evaluated, and to continue code execution with the appropriate alternatives:

MKPATFRAME takes l entries off the top of stack W and puts them in the same order into an A-frame created in the heap, with l read from the topmost entry in R.

JPGUARD *pp* immediately follows the code of a guard term to continue code execution either at the next instruction in sequence or, if the guard evaluates to `false`, at the code referenced by the pointer *pp*. Before branching to *pp*, the instruction clears stack R in the same way as JPFAIL, and it also releases the A-frame whose pointer is topmost on stack A.

DROP_R is the last instruction of a pattern match that complements INIT_R: upon a successful match, it removes from stack R the remaining index tuple and the dummy pointer.

POP_W pops the topmost item off stack W.

It takes just one more instruction to deal with the simple wild card patterns that we have included:

MKRESTLIST puts the remaining elements of a (sub)list, starting at the current traversal position, into a list created in the heap (which may be empty if no more elements are left).

Permitting wild cards between any syntactical positions within a pattern (or subpattern) would complicate matters considerably as it would require more instructions, e.g., to test for the number of elements remaining in a (sub)list, to undo bindings, and to effect backtracking in cases of mismatching (sub)structures.

11.5 Compiling Pattern Matches to LASM Code ∗

Compiling pattern matches specified as preprocessed high-level `case` constructs to abstract machine code follows essentially the same routine as the compilation of ordinary AL programs. It sets out by applying the compilation scheme $\mathcal{C}$ to a λ-lifted `case` function and recursively breaks down as follows:

$$
\begin{aligned}
(1)\ \mathcal{C}\ [\ \widetilde{\mathbf{\Lambda}}_1 \ldots \widetilde{\mathbf{\Lambda}}_s\ \texttt{case}\ \ldots\ pattern_i \parallel guard_i\ \rightarrow\ e_i\ \ldots\ \texttt{otherwise}\ e_0 : es\ ] \\
\Longrightarrow \text{MKFRAME\_T}\ s;\ \text{BRA}\ pp_1;\text{FREE\_T};\ \text{RET};\ \mathcal{C}[\ es\ ]; \\
\qquad \ldots \\
\qquad pp_i\ \rightsquigarrow \mathcal{C}_{\mathcal{T}}\ [\ pattern_i \parallel guard_i, pp_{(i+1)}\ ]\ ;\ \mathcal{C}[\ e_i\ ];\ \text{FREE\_A};\ \text{RET}; \\
\qquad \ldots \\
\qquad pp_0\ \rightsquigarrow\ \text{POP\_W};\ \mathcal{C}\ [\ e_0\ ];\ \text{RET};\ .
\end{aligned}
$$

It generates a top-level instruction sequence that first creates, with the instruction MKFRAME_T s, a T-frame from the topmost s entries on stack W and then branches, via the pointer pp_1, to the code of the first pattern abstraction, thereby also pushing a return continuation onto stack R (instruction BRA pp_1). When returning to this level from a successful match, the T-frame is released by the instruction FREE_T, and the subsequent RET instruction returns control to the code that called the `case` function.

The code for the escape hatch e_0 must be preceded by a POP_W instruction to throw away the argument pointer that is still on stack W, and be followed by RET to return to the top-level `case` code.

Following this piece of code, $\mathcal{C}$ drives the compilation into the individual pattern abstractions, where another compilation scheme $\mathcal{C}_{\mathcal{T}}$ takes over to compile the patterns. The body terms are compiled by $\mathcal{C}$, followed by instructions FREE_A and RET that release the A-frames created after successful matches

and complement the BRA pp_1 instruction of the top-level sequence, respectively. The codes of the individual pattern abstractions are chained together by pointers pp_i that are passed along as parameters of $\mathcal{C}_\mathcal{T}$ to generate, within the pattern code, the branch labels through which, in case of mismatches, control is transferred to the next pattern abstraction in sequence.

The compilation scheme $\mathcal{C}_\mathcal{T}$ distinguishes between the cases where the top-level patterns are variables (whose preprocessed variants are the anonymous binders $\square$), strings, and recursively lists of patterns, and it also compiles the guard terms:

(2) $\mathcal{C}_\mathcal{T}[\ \square \parallel guard,\ pp\ ] \implies$ INIT_R; BIND;
MKPATFRAME; $\mathcal{C}[\ guard\ ]$; JPGUARD pp; DROP_R; POP_W;

(3) $\mathcal{C}_\mathcal{T}[\ `string' \parallel guard,\ pp\ ] \implies$ INIT_R; IS_STRING $`string'$; JPFAIL pp;
MKPATFRAME; $\mathcal{C}[\ guard\ ]$; JPGUARD pp; DROP_R; POP_W;

(4) $\mathcal{C}_\mathcal{T}[\ <^k \ldots > \parallel guard,\ pp\ ] \implies$ INIT_R; $\mathcal{P}_\mathcal{T}[\ <^k \ldots >,\ pp\ ]$;
MKPATFRAME; $\mathcal{C}[\ guard\ ]$; JPGUARD pp; DROP_R; POP_W;

In all three cases, we have the same piece of trailing code that, assuming a successful pattern match, first creates an A-frame for the instantiations of the pattern variables, followed by the code for the guard term, by a conditional branch to the code pointed to by pp if the guard evaluates to `false`, and by DROP_R that is executed if the guard can be satisfied.

The codes preceding these trailers are rather straightforward. As pattern variables always match a corresponding argument (sub)structure, the code for $\square$ simply binds the pointer to it (which is topmost in R) by pushing it onto stack W. The code that does the string matching branches conditionally to the code pointed to by pp if the match fails, and otherwise simply continues with the trailer code. If the pattern is a list of patterns, then $\mathcal{C}_\mathcal{T}$ calls another compilation scheme $\mathcal{P}_\mathcal{T}$ that takes care of this situation:

(5) $\mathcal{P}_\mathcal{T}[\ <^k pattern_1\ pattern_2\ \ldots >,\ pp] \implies$
IS_LIST k; JPFAIL pp; DOWN; $\mathcal{P}_\mathcal{S}[\ pattern_1 : pattern_2 : \ldots\ >,\ pp\ ]$; .

It generates code that first checks whether the arities of the pattern and of the argument list are the same and, if so, has the pointer to it dereferenced by the instruction DOWN. The code for the sequence of subpatterns is generated by yet another compilation scheme $\mathcal{P}_\mathcal{S}$. This is followed by the instruction UP which returns control to the next higher pattern level.

$\mathcal{P}_\mathcal{S}$ splits up into another five compilation rules:

(6) $\mathcal{P}_\mathcal{S}[\ >,\ pp\ ] \implies$ UP;

(7) $\mathcal{P}_\mathcal{S}[\ \texttt{as}[\ \square\ ]\ >,\ pp\ ] \implies$ MKRESTLIST; UP;

(8) $\mathcal{P}_\mathcal{S}[\ \square : pattern : \ldots,\ pp\ ] \implies$ NEXT; BIND; $\mathcal{P}_\mathcal{S}[\ pattern\ \ldots,\ pp\ ]$;

(9) $\mathcal{P}_{\mathcal{S}}[\ `string' : pattern : \ldots,\ pp\ ] \implies$
NEXT; IS_STRING *'string'*; JPFAIL *pp*; $\mathcal{P}_{\mathcal{S}}[\ pattern\ \ldots,\ pp\ ]$;

(10) $\mathcal{P}_{\mathcal{S}}[\ <^k\ \ldots\ > : pattern : \ldots,\ pp\ ] \implies$
NEXT; $\mathcal{P}_{\mathcal{T}}[\ <^k\ \ldots\ >,\ pp\ ]$; $\mathcal{P}_{\mathcal{S}}[\ pattern :\ \ldots\ ,\ pp\ ]$; .

When applied to an empty sequence of subpatterns, identified by the end-of-list symbol >, it inserts an UP instruction (rule (6)) that returns control to the next higher pattern level, a wild card (which must be followed by >) compiles to MKRESTLIST; UP;... (rule (7)), and rules (8), (9), (10) compile variables, strings and sublists of patterns in essentially the same way as rules (2), (3), (4), respectively.

11.6 Code Generation and Execution: an Example **

To illustrate how pattern compilation works, we consider as an example the λ-lifted and preprocessed `case` construct of p. 259 that includes two pattern abstractions. Applying the compilation scheme $\mathcal{C}$ to it (rule (1)) gives the following intermediate code:

pp_{case} $\rightsquigarrow$ MKFRAME_T 2; BRA pp_1; FREE_T; RET;

pp_1 $\rightsquigarrow$ $\mathcal{C}_{\mathcal{T}}[\ <^2\ \square\ \square\ >\ \|\ (\texttt{eq}\ \#1\ \sim 1),\ pp_2\ ]$;
$\mathcal{C}[\ <<\ \#1\ \sim 0\ ><\ \#0\ \sim 1\ >>\ ]$; FREE_A; RET;
pp_2 $\rightsquigarrow$ $\mathcal{C}_{\mathcal{T}}[\ <^2\ \square\ \ <^2\ \square\ \square\ >>\ \|\ (\texttt{gt}\ \#1\ \sim 1),\ pp_0\ ]$;
$\mathcal{C}[\ <\ \#0\ <\ \#2\ \#0\ >\ \sim 1\ >\ ]$; FREE_A; RET;
pp_0 $\rightsquigarrow$ POP_W; $\mathcal{C}[\ <\sim 0\ \sim 1\ >\ ]$; RET; .

This code has the compilation scheme $\mathcal{C}_{\mathcal{T}}$ driven in front of the patterns which, through rule (4), repeatedly calls the compilation schemes $\mathcal{P}_{\mathcal{T}}$ (rule (5)) and $\mathcal{P}_{\mathcal{S}}$ (rule (10)) to yield

pp_{case} $\rightsquigarrow$ MKFRAME_T 2; BRA pp_1; FREE_T; RET;
pp_1 $\rightsquigarrow$ INIT_R; IS_LIST 2; JPFAIL pp_2; DOWN;
NEXT; BIND; NEXT; BIND; UP;
MKPATFRAME; $\mathcal{C}[\ (\texttt{eq}\ \#1\ \sim 1)\ ]$; JPGUARD pp_2; DROP_R; POP_W;
$\mathcal{C}[\ <<\ \#1\ \sim 0\ ><\ \#0\ \sim 1\ >>\ ]$; FREE_A; RET;
pp_2 $\rightsquigarrow$ INIT_R; IS_LIST 2; JPFAIL pp_0; DOWN; NEXT; BIND; NEXT;
IS_LIST 2; JPFAIL pp_0; DOWN;
NEXT; BIND; NEXT; BIND; UP; UP;
MKPATFRAME; $\mathcal{C}[\ (\texttt{gt}\ \#1\ \sim 1)\ ]$; JPGUARD pp_0; DROP_R; POP_W;
$\mathcal{C}[\ <\ \#0\ <\ \#2\ \#0\ >\ \sim 1\ >\ ]$; FREE_A; RET;
pp_0 $\rightsquigarrow$ POP_W; $\mathcal{C}[\ <\sim 0\ \sim 1\ >\ ]$; RET; .

The remaining applications of the compilation scheme $\mathcal{C}$ to the guard and body expressions of both pattern abstractions and to the `otherwise` expression follow the $\mathcal{C}$-rules of Fig. 10.6. They produce the following pieces of code:

$$\mathcal{C}[\,(\texttt{eq}\ \#1\ \sim 1)\,] \Longrightarrow \text{COPY\_TW } 1;\ \text{COPY\_AW } 1;\ \text{REDUCE } 1;\ \text{UPDATE } 1;\ \text{REDUCE } 0;\ \text{UPDATE } 0;\ \text{PUSH\_FUN } \texttt{eq};\ \text{AP } 2;$$

$$\mathcal{C}[\,(\texttt{gt}\ \#1\ \sim 1)\,] \Longrightarrow \text{COPY\_TW } 1;\ \text{COPY\_AW } 1;\ \text{REDUCE } 1;\ \text{UPDATE } 1;\ \text{REDUCE } 0;\ \text{UPDATE } 0;\ \text{PUSH\_FUN } \texttt{gt};\ \text{AP } 2;$$

$$\mathcal{C}[\,<\sim 0\ \sim 1>\,] \Longrightarrow \text{PUSH\_W } pp_{l1};\quad pp_{l1} \rightsquigarrow <\text{COPY\_TW } 0\quad \text{COPY\_TW } 1>$$

$$\mathcal{C}[\,<<\#1\ \sim 0><\#0\ \sim 1>>\,] \Longrightarrow \text{PUSH\_P } pp_{l2};\quad pp_{l2} \rightsquigarrow <\mathcal{C}[\,<\#1\ \sim 0>\,]\ \mathcal{C}[\,<\#0\ \sim 1>\,]>$$

$$\mathcal{C}[\,<\#0\ <\#2\ \#0>\sim 1>\,] \Longrightarrow \text{PUSH\_P } pp_{l3};\quad pp_{l3} \rightsquigarrow <\text{COPY\_AW } 0\ \ \mathcal{C}[\,<\#2\ \#0>\,]\quad \text{COPY\_TW } 1> \quad .$$

To see how this code executes, we consider as an example the application of the `case` function to an argument list $<4\ <3\ 2>>$ and to instantiations of the free variables w and z with the values 3 and 5, respectively, i.e., we have in λ-lifted AL notation:

$$((\tilde{\ }\ \tilde{\lambda}wz.\texttt{case}\ldots\texttt{otherwise}\ <w\ z>\ 3\ 5\,)\ <4\ <3\ 2>>) \quad .$$

Figure 11.3 shows the sequence of instructions executed and the effects that they have on the stacks W and R and on the heap H. The sequence begins with an initial stack configuration that has in stack W the arguments of the tilde application sitting on top of the pointer p_l to the argument list in the heap. As its second element, this list includes another pointer p_{ll} to the sublist $<3\ 2>$:

$$p_l \rightsquigarrow <4\ p_{ll}> \quad p_{ll} \rightsquigarrow <3\ 2> \quad .$$

Stack R is still empty with regard to items that belong to the `case` application.

The `case` code is entered through the pointer pp_{case} to have in step 1 the entries 3 and 5 taken off stack W to create a T-frame, leaving just the argument pointer p_l in W. The BRA pp_1 instruction of step 2 branches to the code of the first pattern abstraction and pushes the remainder of the top-level code, beginning with the instruction FREE_T, as a return continuation onto stack R. The INIT_R instruction of step 3 initializes stack R for the pattern match by pushing, in this order, the dummy pointer $\diamond$ as a separation symbol, an index tuple $(0, 0)$ associated with $\diamond$, and the argument pointer p_l (which it copies from stack W).

Steps 4 to 11 perform the pattern match. After the top-level argument list has been successfully checked for matching arity, its elements are bound by

STEP	INSTR	STACK W	STACK R
0	*init*	$3\ 5\ p_l\ \ldots$	$\ldots$
1	MKFRAME_T	$p_l\ \ldots$	$\ldots$
	creates in heap a T-frame $p_T \rightsquigarrow <\ 3\ 5\ >: H \mid$ p_T is pushed onto stack T		
2	BRA pp_1	$p_l\ \ldots$	FREE_T$\ldots$
3	INIT_R	$p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
4	IS_LIST 2	`succ` $p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
5	JPFAIL pp_2	$p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
6	DOWN	$p_l\ \ldots$	$(0,0)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
7	NEXT	$p_l\ \ldots$	$4\ (1,0)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
8	BIND	$4\ p_l\ \ldots$	$(1,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
9	NEXT	$4\ p_l\ \ldots$	$p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
10	BIND	$p_{ll}\ 4\ p_l\ \ldots$	$(2,2)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
11	UP	$p_{ll}\ 4\ p_l\ \ldots$	$(0,2) \diamond$ FREE_T$\ldots$
12	MKPATFRAME	$p_l\ \ldots$	$(0,2) \diamond$ FREE_T$\ldots$
	creates an A-frame $p_A \rightsquigarrow <\ p_{ll}\ 4\ >: H \mid$ p_A is pushed onto stack A		
13	$\mathcal{C}$[(`eq` #1 $\sim$ 1)]	`false` $p_l\ \ldots$	$(0,2) \diamond$ FREE_T$\ldots$
14	JPGUARD pp_2	$p_l\ \ldots$	FREE_T$\ldots$
	releases the A-frame again by popping p_A off stack A		
15	INIT_R	$p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
16	IS_LIST 2	`succ` $p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
17	JPFAIL pp_0	$p_l\ \ldots$	$p_l\ (0,0) \diamond$ FREE_T$\ldots$
18	DOWN	$p_l\ \ldots$	$(0,0)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
19	NEXT	$p_l\ \ldots$	$4\ (1,0)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
20	BIND	$4\ p_l\ \ldots$	$(1,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
21	NEXT	$4\ p_l\ \ldots$	$p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
22	IS_LIST 2	`succ` $4\ p_l\ \ldots$	$p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
23	JPFAIL pp_0	$4\ p_l\ \ldots$	$p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
24	DOWN	$4\ p_l\ \ldots$	$(0,0)\ p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
25	NEXT	$4\ p_l\ \ldots$	$3\ (1,0)\ p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
26	BIND	$3\ 4\ p_l\ \ldots$	$(1,1)\ p_{ll}\ (1,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
27	NEXT	$3\ 4\ p_l\ \ldots$	$2\ (2,1)\ p_{ll}\ (2,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
28	BIND	$2\ 3\ 4\ p_l\ \ldots$	$(2,2)\ p_{ll}\ (1,1)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
29	UP	$2\ 3\ 4\ p_l\ \ldots$	$(1,3)\ p_l\ (0,0) \diamond$ FREE_T$\ldots$
30	UP	$2\ 3\ 4\ p_l\ \ldots$	$(0,3) \diamond$ FREE_T$\ldots$
31	MKPATFRAME	$p_l\ \ldots$	$(0,3) \diamond$ FREE_T$\ldots$
	creates an A-frame $pp_A \rightsquigarrow <\ 2\ 3\ 4\ >: H \mid$ pp_A is pushed onto stack A		
32	$\mathcal{C}$[(`gt` #1 $\sim$ 1)]	`true` $p_l\ \ldots$	$(0,3) \diamond$ FREE_T$\ldots$
33	JPGUARD pp_0	$p_l\ \ldots$	$(0,3) \diamond$ FREE_T$\ldots$
34	DROP_R pp_0	$p_l\ \ldots$	FREE_T$\ldots$
35	POP_W	$\ldots$	FREE_T$\ldots$
	$\ldots$		

Fig. 11.3. Execution of the compiled `case` code as on the opposite page

pushing them onto stack W, thereby also advancing the traversal position i and the binding index l of the tuple that is on top of stack R. The instruction

UP of step 11 then removes the top level-index tuple and the argument pointer from R, adding the binding index $l = 2$ to the index l of the index tuple that now pops to the top.

The MKPATFRAME instruction of step 12 takes this index to create an A-frame from two entries that are popped off stack W.

Step 13 summarizes the code execution of the guard term. It compares the second entries (with indices #1 and ~ 1) of the A- and T-frames, finds them not to be identical, and pushes as the result the value `false` onto stack W. The instruction JPGUARD of step 14 thereupon pops the `false` entry, clears stack R down to and including the dummy pointer $\diamond$, releases the A-frame that is topmost in stack A and branches to the code of the second pattern abstraction.

Steps 15 to 30 go through essentially the same motions again, this time over both nesting levels of the argument, to produce another match.

The instruction MKPATFRAME of step 31 takes three entries off stack W to create another A-frame in the heap, and the subsequent code of the guard term returns `true` since the second entry of the T-frame (with index ~ 1) is greater than the second entry of the A-frame (with index #1), whereupon in steps 34 and 35 stack R is cleared down to and including $\diamond$, and the argument pointer p_l is finally popped off stack W to turn control over to the code that computes the body of the second pattern abstraction (which is not made explicit in the figure).

11.7 Summary

This chapter is on pattern matching – an operation that extracts substructures from structural contexts and substitutes them for placeholders in other structural contexts. Its implementation blends nicely into the LASM variant of the π–RED machines described in the preceding chapter.

Pattern matching may be specified in an AL-compatible fashion as an ordered set of pattern abstractions collected in a `case` construct that realizes a unary function. Each pattern abstraction is composed of a pattern that is usually specified as a nested list of variables and character strings (including numbers), followed by a guard term and a body expression. The pattern abstractions are tried on the argument in the order in which they are written down. The first matching pattern, including a satisfiable guard term, is the one that succeeds. It returns as value of the entire `case` application the value of the abstraction body in which all occurrences of pattern-bound variables are substituted by the corresponding (sub)structures of the argument.

Though basically applicable to list structures only, this pattern-matching concept includes the notational means to specify customized constructor terms that render it possible to quickly specify (meta-)languages that sit on top of AL proper and to prototype compilers, type systems or interpreters (abstract machines) for them, or to implement term rewrite systems and theorem provers.

Performing pattern matches in the LASM takes a few more instructions to traverse a given argument along the structure specified by the pattern to decide, at each traversal position, whether or not a match occurs or some (sub)structure must be bound to a pattern variable, and to undo all bindings accumulated so far before navigating, in case of a mismatch, to the next pattern abstraction in sequence. Other than that, pattern abstractions are executed in the same way as ordinary λ-abstractions: instantiations of nonlocal variables are held in a T-frame created upon entry into the code of the surrounding `case` function, and instantiations of pattern variables are held in A-frames created upon the successful completion of pattern matches. The codes for the guard and body terms of the pattern abstractions are generated by the very same compilation scheme $\mathcal{C}$ as used in the LASM.

References

Pattern matching is available in all function-based and functional languages, e.g., in SML [Pau96, Ull98, HaRi99], MIRANDA [Tur85], CLEAN [PvE93] and HASKELL [Hudall92, Tho96]. It is also an integral part of the implementation of logic-based languages such as LOGLISP [RoSi82] or PROLOG [Cam84, AK91]. A good account of pattern matching and how it works is given in the textbooks by Kogge [Kog91], Plasmeijer and vanEekelen [PvE93], and earlier by Peyton Jones [PeyJ87], which also includes a chapter by Wadler on compiling pattern matches. Pattern matching is also addressed in a more recent textbook by Baader and Nipkov on term rewriting [BaNi99]. The style of programming with pattern matches that has been used throughout this chapter has been adopted from [Kge94].

12

Another Functional Abstract Machine

We will now study another weakly normalizing λ-calculus machine that brings us one step closer to abstract machines that implement conventional imperative languages. It derives more or less directly from the original SECD machine of Sect. 5.1 but replaces direct interpretation of λ-expressions by execution of abstract instructions. This SECD_I machine (the _I stands for instruction-based) is called functional since it evaluates expressions to values (their weak normal forms), and the evaluation is referentially transparent and confluent, meaning that the value of a (sub)expression is invariant against the context in which it occurs and against evaluation orders. The machine implements an applicative order (or strict) reduction strategy that corresponds to the call-by-value semantics of almost all imperative languages. In contrast to the SASM of Sect. 10.4, which also normalizes weakly under an applicative order regime, the SECD_I machine abandons the concept of closed contexts and instead works with open abstractions, closures, and nested runtime structures composed of linked frames, very similar to the G_HOR machine and the B-machine of Chaps. 7 and 8, respectively, but with a naive substitution mechanism.

The SECD_I machine is a perfect target for the compilation of AL (see Chap. 3) but also for such well-known function-based languages as Standard ML and SCHEME[1] (a purified variant of LISP) that come with an applicative order semantics as well. Its instruction set and operating principles are very similar to those of Cardelli's functional abstract machine (FAM). Many of the instructions are shared with the code-executing abstract machines of Chaps. 8, 9 and 10, and so are parts of the compilation scheme that translates AL expressions into SECD_I code.

[1] In fact, AL expressions may be converted rather straightforwardly into SCHEME expressions, basically by turning defining equations $f = e$ and abstractions `lambda` $u_1 \ldots u_n$ `in` e into the parenthisized forms $(f\, e)$ and (`lambda` $(\, u_1 \ldots u_n\,)\, e$), respectively, and n-ary lists must in SCHEME be represented as quoted expressions $(e_1 \ldots e_n)$.

Compilation is a one-way affair preceded by a preprocessing phase that converts `lambda-bound variables` into index representations for direct translation into environment accesses. There is no complementary decompilation followed by postprocessing that, as in the π-RED machines, would convert terminal machine states into intelligible output other than ground terms such as (lists of) numbers, Boolean values or character strings denoting function results or nonreducible applications.

12.1 The Machine and How It Basically Works

The SECD_I machine centers around an addressable memory that is partitioned into three nonoverlapping sections C, E and H. These sections hold the executable program code, the runtime environment and the heap, respectively. There is an explicit pointer pc – the program counter – to the instruction in C that is being executed, and an explicit environment pointer pe to the topmost frame of the current environment in E. The machine also has a value stack S and a dump stack D for return continuations. A complete machine state is thus described by a tuple

$$(pc,\ pe,\ S,\ E,\ C,\ D,\ H)\ .$$

$C[\ pc \rightsquigarrow instr\ ]$, $E[\ pe \rightsquigarrow frame\ ]$ and $H[\ pp \rightsquigarrow object\ ]$ respectively denote an instruction in C referenced by pc, a frame in C referenced by pe, and a heap object referenced in H by a pointer pp.

The pointers are in fact memory locations given as integer numbers from an interval $[\ 0 \ldots s-1\ ]$, where s is the size of the particular memory section (which is either E, C or H). For the sake of simplicity we assume that all instructions in C have unit length so that two consecutive instructions $instr\_i$ and $instr\_(i+1)$ can be found in memory locations pc_i and pc_i+1, respectively. Likewise, if pe points to a particular environment frame, then the i-th entry of this frame is located at the address $pe+i$. Environment frames (which occupy as many consecutive locations as there are entries) are dynamically allocated in E as space can be made available; the same applies to the placement in H of heap objects of varying size (of which those that are of primary interest here are closures).

Code execution exactly mimics the control mechanism of conventional computing machines: the code is traversed by advancing the program counter pc step by step in increments of one, unless it is updated by a branch instruction to point to some instruction other than the next one in sequence. Primitive value-transforming instructions operate exclusively on the value stack S, which is also used to prepare the arguments for function calls. Branching to function code includes saving, in the form of a tuple $(pc,\ pe)$, the return continuation in the dump stack D. As its first action, the code of the called function creates from the entries deposited by the calling function in S the argument frame it has to work with, prepends it to the calling environment in E

identified by the current pointer *pe*, and uses the pointer to this new frame as the new *pe*. Conversely, returning from a function call may be accomplished simply by retrieving from the dump the topmost return continuation, which implicitly also releases from the current environment the argument frame when the pointer to it (which is held in *pe*) is overwritten.[2]

12.1.1 Some Semantic Issues

We will now introduce some semantic restrictions that are aimed at simplifying the SECD_I machinery, as compared with the SASM of Sect. 10.4, and, in consequence, the compilation of high-level programs to SECD_I code.

With regard to function calls, we demand that they be executable only if the function's arity matches the number of arguments supplied, and that otherwise the computation be terminated with some string value denoting a mismatch. In terms of AL semantics as defined in section 3.2, this behavior may be exemplified by means of a `let` expression as

$$\text{EVAL}[\![\ \texttt{let}\ f = \texttt{lambda}\ v_1\ \ldots\ v_n\ \texttt{in}\ e_0\ \texttt{in}\ (f\ e_1 \ldots e_m)\ ]\!] = \begin{cases} \text{EVAL}[\![\ e_0[\ v_1 \leftarrow \text{EVAL}[\![\ e_1\ ]\!] \ldots v_m \leftarrow \text{EVAL}[\![\ e_m\ ]\!]\ ] & \text{if } n = m, \\ \text{"function } f \text{ of arity } n \text{ receiving } m \text{ arguments"} & \text{otherwise .} \end{cases}$$

The string value returned in the case of mismatching arities becomes the value of the entire program expression regardless of the nesting level at which the mismatch occurs, i.e., it pops to top level.

This semantics ensures that whenever the code of an abstraction is called, it finds exactly the right number of arguments in the right places of an argument frame; otherwise, code execution stops prematurely at the calling instruction.

Also, weak normalization defines the values of unapplied abstractions to be closures that associate the abstractions as they are to the environments that instantiate their relatively free (or nonlocal) variables. However, unapplied abstractions are not decompiled into high-level output but simply turned into the anonymous string value "function", i.e., we have

$$\text{EVAL}[\![\ \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ ]\!] = \text{"function"} ,$$

if $\texttt{lambda}\, v_1 \ldots v_n\ \texttt{in}\, e_0$ either is or, in the course of reducing a program expression, becomes the top-level expression, or it occurs as operand of a top-level application that has the general form $(e_0 \ldots \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_i \ldots)$, where

[2] Of course, the frame itself continues to exist in E, from where it may be garbage-collected as soon as there exist no more references to it anywhere else, e.g., in closures.

e_0 is something other than an abstraction; otherwise, EVAL reproduces the abstraction as it is.

Apart from minor syntactical differences regarding the resulting string values, this is how almost all implementations of functional (or function-based) languages treat applications of abstractions with mismatching arities and unapplied abstractions in order to get around the problems of evaluating closures (which may include the resolution of potential name clashes) and of decompiling function code.

To illustrate what happens when this semantics is used, consider the `let` expression

$$\texttt{let}\ f = \texttt{lambda}\ u\ v\ w\ \texttt{in}\ (u\ (v\ w\ w)\ w)\ \texttt{in}\ e_0\ ,$$

where f represents an abstraction of arity 3. It evaluates to[3]

- 6 if $e_0 =_s (f\ +\ +\ 2)$ since f is applied to the correct number of arguments;
- "function f of arity 3 receiving 2 arguments" if $e_0 =_s (f\ +\ +)$ since one argument is missing, i.e., we have a partial application;
- "function f of arity 3 receiving 4 arguments" if $e_0 =_s (f\ +\ +\ 2\ 3)$ since there is one argument too many;
- "function" if $e_0 =_s f$ since the abstraction f remains unapplied.

This semantics seems to rule out programming techniques that make use of partial applications. However, we can work around this problem by switching to curried representations of abstractions in conjunction with corresponding nestings of applications.

For instance, if the above `let` expression is changed to

$$\texttt{let}\ f_c = \texttt{lambda}\ u\ \texttt{in}\ \texttt{lambda}\ v\ \texttt{in}\ \texttt{lambda}\ w\ \texttt{in}\ (u\ (v\ w\ w)\ w)\ \texttt{in}\ e_0\ ,$$

it evaluates to

- 6 again if $e_0 =_s (((f_c\ +)\ +)\ 2)$ since each of the nested unary abstractions is applied to just one argument;
- "function f_c of arity 1 receiving 2 arguments" if $e_0 =_s ((f_c\ +\ +)\ 2)$ since the outermost unary abstraction is applied to two arguments;
- "function f_c of arity 1 receiving 2 arguments" if $e_0 =_s ((f_c\ +)\ +\ 2)$ since the next to outermost unary abstraction is applied to two arguments (the outermost abstraction is applied correctly);
- "function" if $e_0 =_s (f_c\ +)$ or $e_0 =_s ((f_c\ +)\ +)$ since the next to outermost or the innermost abstraction, respectively, remains unapplied, i.e., we have the equivalent of partial applications.

Generally, we can have curried abstractions of any arity, as, for instance, in the `let` expression

$$\texttt{let}\ h = \texttt{lambda}\ u_1\ u_2\ \texttt{in}\ \texttt{lambda}\ v_1\ v_2\ v_3\ \texttt{in}\ \texttt{lambda}\ w_1\ \texttt{in}\ e_b\ \texttt{in}\ e_0\ .$$

[3] The notation used for the string values closely resembles that of the SCHEME variant DrScheme [PLTS96].

It returns the value of the instantiated abstraction body expression e_b if $e_0 = (((h\ a_1\ a_2)\ b_1\ b_2\ b_3)\ c_1)$, the value "function" if one, two or all three of these applications are dropped from outermost to innermost, and the value "function h of arity n receiving m arguments" otherwise.

A more interesting example that involves applications of functions to functions to compute (the equivalent of) new functions has already been extensively discussed in Sect. 2.1. It concerns applications of the functions *twice* and *square*, which in AL notation may be specified as

$$\begin{array}{l}\texttt{let}\\ \quad twice = \texttt{lambda}\ f\ \texttt{in}\ \texttt{lambda}\ u\ \texttt{in}\ (f\ (f\ u))\\ \quad square = \texttt{lambda}\ n\ \texttt{in}\ (*\ n\ n)\\ \texttt{in}\ e_0\ \ ,\end{array}$$

where *twice* is now specified as a curried function. This expression produces the value "function" if, for instance, $e_0 =_s (twice\ square)$ or $e_0 =_s (twice\ twice)$ or $e_0 =_s ((twice\ twice)\ square)$ and the value 65536 if $e_0 =_s (((twice\ twice)\ square)\ 2)$.

12.1.2 Index Tuples and the Runtime Environment

To support in our SECD_I machine the semantics as just defined, we have to have a closer look at how we wish to internally represent binding structures and, once we have settled this issue, how we go about constructing and accessing the runtime environment.

To convey a basic idea of what needs to be done here, we consider as an example the following AL expression that is composed of two nested `letrec`s:

$$\begin{array}{l}\texttt{letrec}\\ f = \texttt{lambda}\ u_1\ u_2\ \texttt{in}\\ \qquad \texttt{letrec}\\ \qquad\ g = \texttt{lambda}\ v_1\ v_2\ \texttt{in}\\ \qquad\qquad \texttt{lambda}\ w_1\ w_2\ \texttt{in} \dashv \ldots (\ f\ v_1\ w_2\ ) \ldots (\ (\ g\ u_2\ w_1\ )\ u_1\ v_2\ ) \ldots \vdash\\ \qquad \texttt{in}\ \dashv \ldots (\ (\ g\ u_1\ u_1\ )\ u_2\ u_2\ ) \ldots (\ f\ u_1\ u_2\ ) \ldots \vdash\\ \texttt{in}\ \dashv \ldots (\ f\ 2\ 3\ ) \ldots (\ f\ 3\ 2\ ) \ldots \vdash\ \ .\end{array}$$

This expression defines two functions, of which g is local to f. They are called in the body expressions (following the key word `in`) of their defining `letrec`s, recursively by themselves, and g also calls f. The formal parameters of the functions have been chosen so that we have both local and nonlocal bindings, i.e., the variables u_1 and u_2 bound in f occur (relatively) free in g.

The nesting of function definitions requires that a call of f must execute in an empty environment, and that a call of g must execute in an environment created by an application of f. Assuming that each `lambda` binder creates another environment frame, we recognize that when f is called inside g, the

environment for f may be found a distance of three intervening frames away from the the topmost frame of the current environment. These are the frames in which the `lambda`-bound variables (or parameters) $u_1,\ u_2$ of f and $v_1,\ v_2$ as well as $w_1,\ w_2$ of g are instantiated. Similarly, when f is called in f or g in g, the respective environment is a distance of one or two intervening frames away from the top of the current environment, and when f or g are called in the bodies of their defining `letrec`s, the distances are zero.

This is to say that each individual occurrence of a function identifier, say foo, can in an equivalent Λ-expression obviously be represented by a tuple of the form [$f\_ind\ foo$], in which the index f_ind specifies the distance, in terms of intervenening `lambda`s, to the `lambda` that binds the function parameters. We will therefore refer to this index as the lambda binding index or the distance index.

In particular, occurrences of variables (or function identifiers) foo that are `letrec`-bound, as in

$$\texttt{letrec}\ \ldots\ foo = \texttt{lambda}\ u_1\ \ldots\ u_n\ \texttt{in}\ e_b\ \ldots\ \texttt{in}\ e_0\ \ ,$$

can be transformed into tuples

- [0 foo] if foo occurs free in the body expression e_0 of the defining `letrec`, with $f\_ind = 0$ indicating that this occurrence is outside the scope of the abstractor `lambda` $u_1\ \ldots\ u_n$;
- [$f\_ind\ foo$] with $f\_ind > 1$ if foo occurs inside e_b at a distance of f_ind intervening `lambda`s away from `lambda` $u_1\ \ldots\ u_n$.

Occurrences of tuples [$f\_ind\ foo$] in expressions must then be given the following interpretation:

- [0 foo] must link up to the environment of the calling function (or to the environment of the defining `letrec`);
- [$f\_ind\ \ foo$] with $f\_ind\ >\ 1$ must be evaluated in the environment obtained by descending f_ind levels down into the current environment (which may be obtained by dereferencing the current environment pointer f_ind times).

This can be expressed by means of a function

$$p'_E = deref(\ f\_ind,\ p_E\ )$$

that takes the current environment pointer p_E and an index f_ind to return the desired environment pointer p'_E.

A similar approach may be taken with `lambda`-bound variables as well, using the following convention: binders `lambda` $u_1\ \ldots\ u_n$ are, as in the abstract machines of the preceding chapters, converted into sequences of n nameless binders $\underbrace{\Lambda\ \ldots\ \Lambda}_{n}$, and occurrences of a bound variable u_k are replaced by index tuples [$i\ j$] rather than by plain binding indices such that

- i denotes the nesting level, or the binding distance, of the occurrence of $[\ i\ j\ ]$ relative to the binding Λ-sequence, given by the number of intervening Λ-sequences;
- j denotes the declaration position of the binder Λ within the binding Λ-sequence, in terms of the number of intervening Λs in this sequence.

For instance, an expression

$$\begin{array}{l}\texttt{lambda}_i\ u_{(n-1)} \ldots u_j \ldots u_0\ \texttt{in} \\ \qquad\qquad \dashv \ldots \texttt{lambda}_0\ v_{(m-1)} \ldots v_0\ \texttt{in}\ \dashv\ \ldots\ u_j\ \ldots \vdash\ \ldots \vdash\end{array}$$

in which we assume to have i intervening `lambda`s between the u_j that is exposed in the body of the inner abstraction and the binding `lambda` translates into

$$\Lambda_{(n-1)} \ldots \Lambda_j \ldots \Lambda_0.\ \dashv\ \ldots\ \Lambda_{(m-1)} \ldots \Lambda_0.\ \dashv\ \ldots [\ i\ j\ ] \ldots \vdash\ \ldots \vdash\ .$$

When converting into this index tuple representation a curried `lambda`-abstraction such as

$$\begin{array}{l}\texttt{lambda}\ u_1\ u_2\ \texttt{in}\ \texttt{lambda}\ v_1\ v_2\ \texttt{in}\ \texttt{lambda}\ w\ \texttt{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad \dashv \ldots v_1 \ldots u_1 \ldots w \ldots v_2 \ldots u_2 \ldots \vdash\ ,\end{array}$$

we take the dots '.' to syntactically separate from each other the Λ-sequences that belong to different nesting levels and separate the innermost Λ-sequence from the abstraction body to enable compilation to code that properly reflects the nesting of abstractions. We thus obtain

$$\Lambda\Lambda.\Lambda\Lambda.\Lambda.\ \dashv \ldots [\ 1\ 1\ ] \ldots [\ 2\ 1\ ] \ldots [\ 0\ 0\ ] \ldots [\ 1\ 0\ ] \ldots [\ 2\ 0\ ] \ldots \vdash\ .$$

Here the distance indices 0, 1 and 2 of the index tuples respectively refer to the inner Λs, to the Λs in the middle, and to the outer Λs.

To see how all this works out, consider as an example again the above `letrec` expression. It translates into the index tuple representation

$$\begin{array}{l}\texttt{letrec} \\ f = \Lambda\Lambda. \\ \quad \texttt{letrec} \\ \quad g = \Lambda\Lambda.\Lambda\Lambda. \\ \quad\ \dashv \ldots ([\ 3\ f\ ]\ [\ 1\ 1\ ]\ [\ 0\ 0\ ]) \ldots (([\ 2\ g\ ]\ [\ 2\ 0\ ]\ [\ 0\ 1\ ])\ [\ 2\ 1\ ]\ [\ 1\ 0\ ]) \ldots \vdash \\ \quad \texttt{in}\ \dashv \ldots (([\ 0\ g\ ]\ [\ 0\ 1\ ]\ [\ 0\ 1\ ])\ [\ 0\ 0\ ]\ [\ 0\ 0\ ]) \ldots ([\ 1\ f\ ]\ [\ 0\ 1\ ]\ [\ 0\ 0\ ]) \ldots \vdash \\ \texttt{in}\ \dashv \ldots ([\ 0\ f\ ]\ 2\ 3) \ldots ([\ 0\ f\ ]\ 3\ 2) \ldots \vdash\ .\end{array}$$

A typical environment as it unfolds when evaluating the above expression is depicted in Fig. 12.1. It is composed, from the bottom up, of a single frame for one call of the function f, followed by two frames for calls of the function

g in f, followed by frames for several recursive calls of f out of the body of g, followed by frames for several recursive calls of g out of f. Each frame includes a link pointer to the preceding frame (or to the empty environment) and two slots for argument values (which may also be pointers to nonatomic values such as closures) with which the function parameters are instantiated.[4]

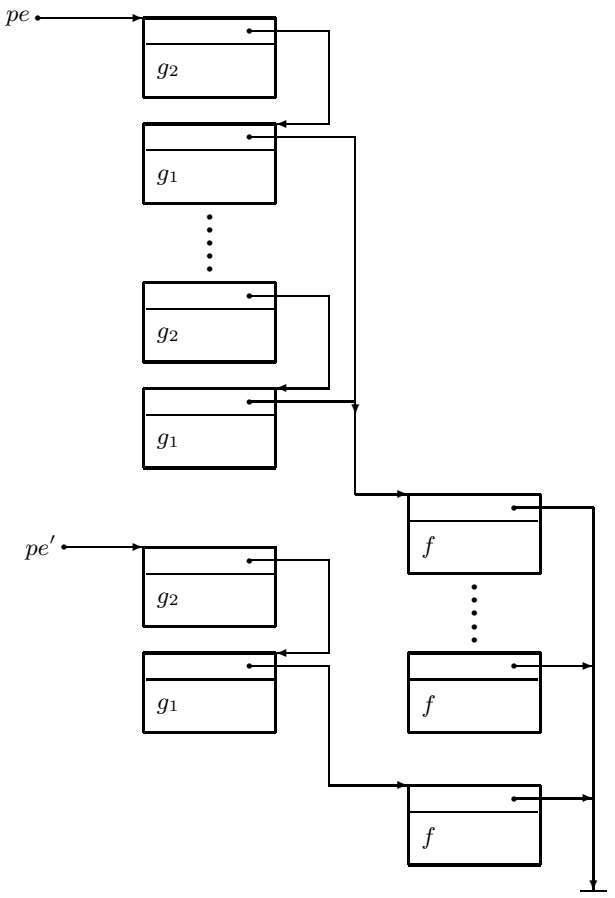


Fig. 12.1. A typical runtime environment of the SECD_I machine

The structure of this environment is basically the same as in Fig. 7.7: it has frames linked up in a cactus-like fashion that reflects the nesting of function definitions. However, since the SECD_I machine is only weakly normalizing, meaning that the semantics of the source language rules out partial applica-

[4] The frames depicted by the boxes are inscribed with the functions that created them, with g_1 and g_2 denoting the frames for the inner and outer applications, respectively, of the function g.

tions other than in the disguise of curried functions and function applications, accessing the environment becomes decidedly simpler than in the fully normalizing machines. The index tuple $[\ i\ j\ ]$ directly specifies, by means of the distance index i, how many times the current environment pointer pe must be dereferenced to arrive at the correct frame, and the declaration position j specifies the offset relative to the base of the frame at which the entry to be accessed is located. More precisely, and using the function $deref$, an index tuple translates into a pointer to an environment location as

$$[\ i\ j\ ]\ \rightarrow\ deref(\ pe,\ i\ ) + j + 1$$

(the additional offset of one is due to the first entry of a frame being reserved for the link pointer to the preceding frame).

We may convince ourselves that the index tuples $[\ f\_ind\ ff\ ]$ and $[\ i\ j\ ]$ access the right frames and the right entries therein by looking at the body of the function g of our example program and at the environment of Fig. 12.1, which has the pointer pe pointing to a frame of g_2. The function call of f in the body of g comes with a distance index 3, meaning that the environment of f can be found by dereferencing pe three times, which gets us to the empty environment. Likewise, the call of g in this body comes with a distance index 2 that, when pe is dereferenced twice, correctly identifies the most recent frame of f as its environment. Also, the index tuples $[\ i\ j\ ]$ with distance indices 0, 1 and 2 correctly refer to entries in the most recent frames of g_2, g_1 and f, respectively, which may be validated by relating the index tuples to the `lambda`-bound variables they replace.

12.2 The SECD_I Instruction Set

The instructions that are of primary interest here are again those that handle function calls, parameter passing and the construction of the environment. A formal definition of these instructions in terms of the usual state transition function

$$\tau_{secd\_i} : (pc,\ pe,\ S,\ E,\ C,\ D,\ H) \rightarrow (pc',\ pe',\ S',\ E',\ C,\ D',\ H')$$

is given in Fig. 12.2.[5]

From top to bottom, we have the following instructions:

PUSH_S *atom* pushes an atomic value or a pointer, say to function code or to a closure, onto stack S;

COPY_ES i j accesses the j-th entry in the i-th frame relative to the top of the current environment and copies it on top of stack S;

[5] Note that the code structure C is the only one that never changes since it is a static structure that is traversed by the instruction counter pc.

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{PUSH\_S}\ atom\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe,\ atom : S,\ E,\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ S,\ E[\ deref(\ pe,\ i) \rightsquigarrow\ ppf : se_1 : \ldots : se_{(j+1)} : \ldots se_n\ ]\ C[\ pc \rightsquigarrow \text{COPY\_SC}\ i\ j\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe,\ se_{(j+1)} : S,\ E[],\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ ppf : se_1 : \ldots : se_n : S,\ E,\ C[\ pc \rightsquigarrow \text{MKFRAME}\ n\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe',\ S,\ pe' \rightsquigarrow\ ppf : se_n : \ldots : se_1 : E,\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ true \mid false : S,\ E,\ C[\ pc \rightsquigarrow \text{BRC}\ p\_t\ p\_f\ ],\ D,\ H\ ) \rightarrow (p\_t \mid p\_f,\ pe,\ S,\ E,\ C[],\ (pc+1,\ pe\ ) : D,\ H\ )$$

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{BSR}\ p\_f\ ],\ D,\ H\ ) \rightarrow (p\_f,\ pe,\ S,\ E,\ C[],\ (pc+1,\ pe\ ) : D,\ H\ )$$

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{ARGS}\ n\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe,\ S,\ E,\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ p\_f : S,\ E,\ C[\ pc \rightsquigarrow \text{AP}\ r,\ p\_f \rightsquigarrow \text{ARGS}\ n,\ \ldots\ ],\ D,\ H\ ) \rightarrow \begin{cases} (p\_f,\ pe,\ S,\ E,\ C[],\ (pc+1,\ pe\ ) : D,\ H\ ) & \text{if } n = r \\ (-, -, [\ mm,\ p\_f,\ r\ ], -, -, -, -\ ) & \text{if } n \neq r \end{cases}$$

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{RET}\ ],\ (ppc,\ ppe\ ) : D,\ H\ ) \rightarrow (ppc,\ ppe,\ S,\ E,\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{LINK}\ h\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe,\ deref(\ pe,\ h\ ) : S,\ E,\ C[],\ D,\ H\ )$$

$$(pc,\ pe,\ S,\ E,\ C[\ pc \rightsquigarrow \text{MKCLOS}\ h\ p\_f\ ],\ D,\ H\ ) \rightarrow (pc+1,\ pe,\ p\_clos : S,\ E,\ C[],\ D,\ p\_clos \rightsquigarrow\ [\ deref(\ pe,\ h\ ),\ p\_f\ ] : H\ )$$

$$(pc,\ pe,\ p\_clos : S,\ E,\ C[\ pc \rightsquigarrow \text{AP}\ r,\ p\_f \rightsquigarrow \text{ARGS}\ n,\ \ldots],\ D,\ H[\ p\_clos \rightsquigarrow\ [\ pp,\ p\_f\ ]\ ]\ ) \rightarrow \begin{cases} (p\_f,\ pp,\ S,\ E,\ C[],\ (pc+1,\ pe\ ) : D,\ H[]\ ) & \text{if } n = r \\ (-, -, [\ mm,\ p\_f,\ r\ ], -, -, -, -\ ) & \text{if } n \neq r \end{cases}$$

Fig. 12.2. The definition of the SECD_I machine instruction set

MKFRAME n is expected to find on top of stack S a pointer ppf to an environment and n value entries se_j underneath, from which it creates another environment frame in E (with the order of entries reversed), and returns the pointer pe' to this frame as the new environment pointer;

BRC $p_t\ p_f$ realizes a two-way conditional branch, as typically needed to implement `if_then_else` clauses: depending on the Boolean value found on top of stack S, code execution continues at either the pointer p_t or the pointer p_f, and in the dump it saves as return continuation the tuple

(pc, pe) that becomes effective after having completed the code for either the consequent or the alternative;

BSR p_f realizes a function call: it branches unconditionally to function code and again saves in the dump the return continuation (pc, pe);

ARGS n is a pseudoinstruction that has no effect; it is the very first instruction of some function code that specifies the function's arity;

AP r in conjunction with a pointer p_f to function code on top of stack S branches to this code if the arity given by its first (pseudo)instruction equals the number r of arguments supplied and saves a return continuation in the dump; otherwise, it issues as a value on stack S a triple of the form [mm p_f r] to signify a mismatch and terminates the computation;

AP r in conjunction with a pointer p_clos to a closure on top of stack S branches to the code of the closure's function if its arity equals the number r of arguments supplied, saves a return continuation in the dump D, and takes over from the closure the new environment pointer; otherwise, it issues the same mismatch triple as above;

RET returns from one of the branches of an `if_then_else` clause or from a function call, taking off the top of the dump stack D the program counter pc and the environment pointer pe with which the computation needs to continue;

LINK h dereferences h times the current environment pointer and pushes the pointer thus obtained on top of stack S as the link pointer of a new frame whose entries are assumed to have been stacked up underneath;

MKCLOS h p_f creates in the heap H a closure from a pointer p_f to function code and an environment pointer obtained by dereferencing h times the current environment pointer pe; the pointer to this new closure is pushed on top of stack S.

In addition, there is the usual set of parameterless arithmetic | logic | relational instructions, which take their arguments off and push their result values onto stack S, as in the SE(M)CD machines described in Chap. 5.

We also have ENTRY and EXIT instructions that respectively initialize the runtime structures and in a regular fashion terminate the machine.

A regular terminal state is reached when pc points to an EXIT instruction. This state may have as the weak normal form of a program expression either (a pointer to) a basic value (or ground term) or a pointer to a closure on the value stack S. Basic values are directly taken as output, closures produce as output the string "function". Premature termination with an AP r instruction in the case of mismatching arities leaves a triple [mm p_f r] on stack S (all other components of the machine state become irrelevant) that translates into the output "function f of arity n receiving r arguments", where n is retrieved from the (pseudo)instruction ARGS n to which p_f points.

12.3 Compilation to SECD_I Code

Compiling AL expressions to SECD_I code may be defined again by a top-level compilation scheme

$$\mathcal{C}[\, e : es \,] \implies code[\, e \,]; \mathcal{C}[\, es \,]$$

that applies to some syntactically complete head expression e and to some tail es in which may accumulate expressions to be compiled later on. The expressions are assumed to have been turned into index tuple representation by a preprocessor. The subset of preprocessed AL expressions in the compilation of which we are primarily interested is:

$$\begin{aligned} e = {} & const \mid [\, f\_ind\ ff \,] \mid [\, i\ j \,] \mid \\ & \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\ & (e_0\ e_1\ \ldots\ e_r) \mid \underbrace{\Lambda \ldots \Lambda}_{n}.e_b \mid \\ & \texttt{letrec}\ f_1 = e_1\ \ldots\ f_m = e_m\ \texttt{in}\ e_0 \quad , \end{aligned}$$

i.e., we have to deal with constant values (including pointers to function codes or to closures), occurrences of index tuples for `letrec` defined function identifiers and for `lambda`-bound variables, conditionals, r-ary applications, n-ary abstractions, and `letrec` constructs for mutually recursive function definitions.

The set of compilation rules is given in Fig. 12.3.

The compilation of all expressions that do not directly involve abstractions follows basically the same rules as those of the abstract machines described in Chaps. 9 and 10. Constant values (and pointers) are pushed onto stack S (rule (1)), index tuples translate into instructions that copy environment entries on top of stack S (rule (2)), `if_then_else` clauses compile to code that, depending on the value of the predicate, branches conditionally to the code of either consequent or alternative (rule (3)), applications have their operands evaluated before the operators are applied to them (rule (4)), and `letrec`s compile to code for their body expressions followed by codes for the individual functions that are generated by another compilation scheme $\mathcal{F}$. These codes are referenced by symbolic pointers that, for identification purposes, carry the function identifiers as subscripts (rule (5)).

An abstraction occurring in operator position of an application with matching arity, after the pointer to the current environment has been pushed onto S, has its code called through a BSR instruction (rule (6)). Nonmatching arities are rejected in this context. If an abstraction occurs anywhere else, say in an operand position of an application, it is simply wrapped up in a closure (rule (9)). The code for the abstraction itself is compiled by $\mathcal{F}$.

A function identifier tuple $[\, h\ ff \,]$ for a `letrec`-defined function, when it occurs in operator position of an application, compiles to a call of the function code in the environment given by the frame index h, provided the arities match (rule (7)). A closure is created if such a reference occurs anywhere else (rule (10)). Thus rules (6) and (9) are essentially the same as rules (7) and (10), respectively, except for the environments to which the function calls link up

(1) $\mathcal{C}[\, const : es \,] \implies \text{PUSH\_S}\ const;\ \mathcal{C}[\, es \,];$

(2) $\mathcal{C}[\, [\, i\ j \,] : es \,] \implies \text{COPY\_ES}\ i\ j;\ \mathcal{C}[\, es \,];$

(3) $\mathcal{C}[\, \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : es \,] \implies \mathcal{C}[\, e_0 \,];\ \text{BRC}\ p_t\ p_f;\ \mathcal{C}[\, es \,];$
$p_t \rightsquigarrow \mathcal{C}[\, e_1 \,];\ \text{RET};\ p_f \rightsquigarrow \mathcal{C}[\, e_2 \,];\ \text{RET};$

(4) $\mathcal{C}[\, (\ e_0\ \ldots\ e_{r-1}\ e_r\ ) : es \,] \implies \mathcal{C}[\, e_r \,];\ \mathcal{C}[\, e_{r-1} :\ \ldots\ : e_0 : ap^r : es \,]$

(5) $\mathcal{C}[\, \texttt{letrec}\ f_1 = e_1\ \ldots\ f_m = e_m\ \texttt{in}\ e_0 : es \,] \implies \mathcal{C}[\, e_0 \,];\ \mathcal{C}[\, es \,];$
$p_{f_1} \rightsquigarrow \mathcal{F}[\, e_1 \,];\ \ldots\ ;\ p_{f_m} \rightsquigarrow \mathcal{F}[\, e_m \,];$

(6) $\mathcal{C}[\, \underbrace{\Lambda \ldots \Lambda}_{r}.e_b : ap^r : es \,] \implies \text{LINK}\ 0;\ \text{BSR}\ p_f;\ \mathcal{C}[\, es \,];$
$p_f \rightsquigarrow \mathcal{F}[\, \underbrace{\Lambda \ldots \Lambda}_{r}.e_b \,]$

(7) $\mathcal{C}[\, [\, h\ ff \,] : ap^r : es \,] \mid ff = \underbrace{\Lambda \ldots \Lambda}_{r}.e_b \implies \text{LINK}\ h;\ \text{BSR}\ p_{ff};\ \mathcal{C}[\, es \,];$

(8) $\mathcal{C}[\, e_0 : ap^r : es \,] \implies \mathcal{C}[\, e_0 \,];\ \text{AP}\ r;\ \mathcal{C}[\, es \,]$

(9) $\mathcal{C}[\, \underbrace{\Lambda \ldots \Lambda}_{r}.e_b : es \,] \implies \text{MKCLOS}\ 0\ p_{ff};\ \mathcal{C}[\, es \,];$
$p_{ff} \rightsquigarrow \mathcal{F}[\, \underbrace{\Lambda \ldots \Lambda}_{r}.e_b \,];$

(10) $\mathcal{C}[\, [\, h\ ff \,] : es \,] \implies \text{MKCLOS}\ h\ p_{ff};\ \mathcal{C}[\, es \,];$

(11) $\mathcal{F}[\, e \,] \implies \begin{cases} \text{ARGS}\ \ r;\ \text{MKFRAME}\ r;\ \mathcal{C}[\, e_b \,];\ \text{RET}; & \text{if } e = \underbrace{\Lambda \ldots \Lambda}_{r}.e_b \\ \text{ARGS}\ 0;\ \mathcal{C}[\, e \,];\ \text{RET} & \text{otherwise} \end{cases}$

Fig. 12.3. Compiling an AL kernel to SECD_I code

and for the fact that in the latter cases the function codes are compiled in other contexts.

Applications that have expressions other than abstractions or direct references to them in operator position have these expressions compiled by $\mathcal{C}$, followed by the instruction AP r (rule (8)). This instruction takes care of the general case in which functions of matching or mismatching arity or something other than functions may end up in this position eventually.

Finally, rule (11) defines the compilation scheme $\mathcal{F}$ that enters the game whenever abstractions have to be turned into code. This code begins with the instructions ARGS and MKFRAME, followed by the code compiled by $\mathcal{C}$ for the abstraction body (which may or may not be another abstraction), and terminates with RET to return to the calling code. The special case where $\mathcal{F}$ is applied to something other than an abstraction is, for reasons of consistency with the context in which the code may be called, treated as an abstraction of arity 0.

As an example, we consider again the `letrec` expression on p. 277 to show how the function g compiles. In order to cover both the special cases of matching arities between functions and applications and the general case of unknown function arity, the function body has been slightly modified to

$$g \;=\; \Lambda\Lambda.\Lambda\Lambda.(\,(\,[\,3\;f\,]\,[\,1\;1\,]\,[\,0\;0\,]\,)\,(\,(\,[\,2\;g\,]\,[\,2\;0\,]\,[\,0\;1\,]\,)\,[\,2\;1\,]\,[\,1\;0\,]\,)\,,$$

i.e., the applications of f and g have become the operator and operand of another application that constitutes the body of the inner abstraction of g.

Compiling the entire `letrec` under which the function g is defined calls for rule (5) of Fig. 12.3, which in turn calls rule (10) to apply the scheme $\mathcal{F}$ for generating function code referenced by a newly created pointer p_{g1} as

$$p_{g1} \;\rightsquigarrow\; \mathcal{F}[\,\Lambda\Lambda.\Lambda\Lambda.(\,\ldots\,)\,]\;\;.$$

$\mathcal{F}$ applied to the outer abstraction yields

$$p_{g1} \;\rightsquigarrow \text{ARGS } 2;\ \text{MKFRAME } 2;\ \text{MKCLOS } 0\ p_{g2};\ \text{RET};$$

and behind the newly created pointer p_{g2} another call of $\mathcal{F}$ compiles the inner abstraction to

$$p_{g2} \;\rightsquigarrow \text{ARGS } 2;\ \text{MKFRAME } 2;\ \mathcal{C}[\,(\,\ldots\,)\,];\ \text{RET};\quad .$$

The first step of applying $\mathcal{C}$ to the remaining abstraction body calls for rule (4) that swaps operator and operand of the outermost application to apply $\mathcal{C}$ recursively as

$$\mathcal{C}[\,(\,(\,[\,2\;g\,]\,[\,2\;0\,]\,[\,0\;1\,]\,)\,[\,2\;1\,]\,[\,1\;0\,]\,)\,];\ \mathcal{C}[\,(\,(\,[\,3\;f\,]\,[\,1\;1\,]\,[\,0\;0\,]\,) : ap^1\,]\;\;.$$

Application of rules (4), (7) and (8) to the operand expression generates the code

$$\begin{aligned}&\text{COPY\_SE } 1\ 0;\ \text{COPY\_SE } 2\ 1;\\ &\qquad\text{COPY\_SE } 0\ 1;\ \text{COPY\_SE } 2\ 0;\ \text{LINK } 2;\ \text{BSR } p_{g1};\ \text{AP } 2;\\ &\qquad\qquad\mathcal{C}[\,(\,(\,[\,3\;f\,]\,[\,1\;1\,]\,[\,0\;0\,]\,) : ap^1\,];\quad .\end{aligned}$$

Here we have the interesting case where the application of the outer abstraction of g (to which p_{g1} points) can be implemented as a BSR p_{g1} instruction since the compiler can decide that the arity of the abstraction is the same as that of the application, but the application of the inner abstraction (to which p_{g2} points) must be implemented with the instruction AP 2 since at compile time nothing is known about the arity of the function returned in the operator position (if it is a function at all).

The same applies to the remaining compilation steps: the application of f to two arguments can be compiled to a BSR p_f instruction, using rule (7), whereas it is not known at compile time whether what is returned by this application is a function of arity one, hence the instruction AP 1 that is set up by rule (8) for the outermost application:

COPY_SE 1 0; COPY_SE 2 1;
COPY_SE 0 1; COPY_SE 2 0; LINK 2; BSR p_{g1}; AP 2;
COPY_SE 0 0; COPY_SE 1 1; LINK 3; BSR p_f; AP 1; .

Thus the complete code for the function g looks like this:

p_{g1} $\rightsquigarrow$ ARGS 2; MKFRAME 2; MKCLOS 0 p_{g2}; RET;

p_{g2} $\rightsquigarrow$ ARGS 2; MKFRAME 2;
COPY_SE 1 0; COPY_SE 2 1;
COPY_SE 0 1; COPY_SE 2 0; LINK 2; BSR p_{g1}; AP 2;
COPY_SE 0 0; COPY_SE 1 1; LINK 3; BSR p_f; AP 1; RET; .

To convince ourselves that this code does what we expect it to do, consider the branch instruction BSR p_{g1} in the third line of the code to which p_{g2} points. It is preceded by two COPY_SE instructions and a LINK instruction that push arguments and a link pointer onto stack S. Upon entering the code referenced by p_{g1}, a new frame made up from the three entries just deposited in S is added to the current environment, a closure including this new environment is created for the code referenced by p_{g2}, and the pointer to it is pushed onto stack S. Following this, control returns to the calling code and continues. The instruction to be executed next is AP 2 that takes the closure pointer off the top of stack S, pushes its environment pointer onto S and attempts to apply the function code referenced by the pointer p_{g2} held in the closure. Since the function's arity as specified by the pseudo-instruction ARGS 2 matches the parameter 2 of AP (which also implies that in S there are two valid arguments underneath the environment pointer), execution proceeds correctly with the code referenced by p_{g2}. The first action of this code is to add another frame of two argument entries to the current environment, i.e., we have two frames linked up to each other as depicted in Fig. 12.1 that hold instantiations of the variables that are `lambda`-bound in the function g.

Replacing the parameter of the instruction AP by some value $r \neq 2$ would terminate the computation with the value "function of arity 2 receiving r arguments", any instruction other than AP r following BSR p_{g1} would simply leave the closure pointer on S untouched.

12.4 Summary

The SECD_I machine is a code-executing abstract machine for functional languages with an applicative order (or strict) semantics. The machine is weakly normalizing, meaning that substitutions do not penetrate, and no evaluation takes place in, abstraction bodies. Abstractions are internally represented as closures, and closures that cannot be applied anywhere are converted into the

anonymous output value "function". Abstractions may be curried and may have arities greater than one, but they (or, more precisely, the respective closures) can be applied successfully only to matching numbers of arguments. Applications of closures may return closures as values. Mismatches terminate the computation with a message signifying the cause of the problem. The machine is therefore a perfect vehicle for implementing functional languages such as ML or SCHEME that both feature an applicative order (or strict) semantics and excatly the same treatment of mismatching function applications.

As the name indicates, the machine is more or less a direct descendant of the classical SECD machine of Chap. 5 as it employs the same runtime structures, but uses them in a slightly different form. Very much like a conventional computing machine, it features a program counter *pc*, a pointer *pe* to the current environment, and a dump D that stacks up return continuations of function calls as tuples (*pc*, *pe*). The value stack S, other than accommodating pointers to closures rather than the closures themselves, is basically the same as in the original SECD machine.

The most important instructions of the machine are those that create and apply closures, or – as shortcuts – branch directly to function code, set up argument frames for function calls, link them up to the calling environments and return from function calls.

Compiling high-level functional programs to SECD_I code is preceded by a preprocessing phase that turns occurrences of

- identifiers ff of `letrec`-defined functions into tuples [h ff] that pair these identifiers with indices h identifying the environments in which the function codes need to be executed;
- `lambda`-bound variables into index tuples [i j] of which the distance index i identifies in the current environment the frame and the declaration index j identifies the entry in the frame by which the variable occurrence must be substituted.

The compiler translates these tuples directly into instructions that respectively create closures and copy environment entries on top of stack S.

The differences compared with the SASM of Sect. 10.4, whose kernel is also an strictly evaluating and weakly normalizing code-executing machine, are due primarily to the internal representation of functions (abstractions) and the ensuing construction of the environments. The SASM works with closed contexts resembling supercombinators (which typically are entire `letrec` expressions) and accordingly splits the environments for evaluating abstraction bodies up into individual frames for instantiations of locally and nonlocally bound variables. Accesses to the environment derive from two different sets of binding indices that define static offsets into each of these frames. This contrasts with the SECD_I machine that works with open abstractions wrapped up in closures. The ensuing environment is a linked list of frames of which what is currently the topmost frame holds instantiations of the variables that are locally bound in the function code that is currently executed, and in-

stantiations of the nonlocal variables are spread out over the frames that are underneath. Thus, what is packed into one frame for nonlocals in the SASM may take several linked frames in the SECD_I machine.

References

The SECD_I machine resembles Cardelli and McQueen's functional abstract machine (FAM) [CMQ83] that is a canonical reference for the implementation of functional (or function-based) languages with an applicative order semantics, e.g., standard ML [Car84]. Another code-executing descendant of the SECD machine is Henderson's LISPKIT implementation [Hen80], which also supports laziness by means of DELAY and FORCE instructions that protect and unprotect terms from being evaluated. Other abstract machines with an applicative order (or strict) semantics include the categorial abstract machine (CAM) of Cousineau, Curien and Mauny [CCM85/87], Leroy's Zinc abstract machine (ZAM) [Ler90] (which is a strict variant of Krivine's $\mathcal{K}$-machine), and the SASM of Gaertner and Kluge [GK96]. The CAM and the SASM are also described in Sects. 5.5.2 and 10.4, respectively, of this text.

As functional languages with a strict semantics lend themselves directly to nonsequential execution, they play a key role in dataflow computing, as exemplified by the Manchester dataflow or the MIT MONSOON machines [GuKiWa85, PaCu90]. The former is representative of numerous machines that try to exploit instruction-level concurrency by direct compilation of high-level programs to dataflow graphs.[6] The latter works with sequential threads laid over the dataflow graphs to avoid excessive synchronization overhead by raising the level of granularity and to facilitate pipelined processing. An abstract machine for thread execution is described in [AAM95].

A similar idea of extracting from functional programs multi-threaded control flow that reflects the underlying dataflow structures but eliminates synchronization overhead is pursued in the DATAROL machinery of Amamiya [Ama91].

The relationship between the demand-controlled reduction model and the dataflow model of computation is discussed in a paper by Amamiya [Ama88] and also in [Kge92].

Other successful implementations of functional languages with a strict semantics that are tuned for high-performance number-crunching applications include the single assignment languages SISAL described by Cann in [Ca89, Ca92] and SAC, recently published by Scholz in [Scho03]. SISAL is first compiled to a tailor-made dataflow language that, in turn, is compiled to *C*, SAC is directly compiled to *C*-macros (which are subsequently expanded) to take in both cases full advantage of code optimizations provided by *C*-compilers. A multi-threaded implementation of SAC that easily outperforms equivalent nonsequential HPF implementations is described by Grelck [Gre01, Gre03].

[6] This approach has now been abandoned, largely because of implementation problems that made it fall short of performance expectations.

13

Imperative Abstract Machines

Throughout the preceding chapters we have looked at several abstract machines that evaluate expressions of an applied λ-calculus, a syntactically sugared version of which is our algorithmic language AL of Chap. 3. The differences among these machines basically concern weak versus full normalization of λ-expressions and, related to that, the treatment of partially applied or unapplied abstractions, applicative order (operands-first) versus normal order (operands-when-needed) evaluation, and interpretation versus code execution.

These machines employ environments to delay substitutions of formal by actual function parameters until it becomes necessary to do so. The environments associate λ-bound variables to the values (or representations of values in the form of closures or suspensions) by which they need to be substituted. However, the environment is not visible to the programmer, meaning that she/he has no means available of inspecting and explicitly manipulating it. Conceptually, variables are their own values; they may be substituted by but do not represent other values. Environment entries may only be created, copied and released, but they can never have their values changed. This property guarantees referential transparency since nothing can be done that could have an effect somewhere else. Computations based on this notion of a hidden environment are therefore said to be independent of an explicitly changeable state. They are oriented toward computing the values of expressions rather than effecting state changes, say, to memorize something that could be used in another context later on.

The classical model of imperative programming and program execution constitutes a radical departure from this approach as the essence of computing here is to effect sequences of elementary state (or environment) changes, explicitly specified by the programmer, that transform step by step some initial into a terminal state. In short, programs are executed for their effects (on the environment) rather than for computing values. This concept primarily reflects itself in another perception of variables: they behave like containers (or boxes) whose contents may be modified by assignment statements. Alternatively, we may say that variables represent values, and assignments change

these values. Several such assignments may be made in succession to the same box variables.

Closely related to assignments is the notion of reference parameters. They are used by abstractions called procedures to effect changes in their calling environments. Procedure calls must have these parameters substituted by variables defined in surrounding contexts (which may have to be specifically annotated as pointers to the values they represent). Assignments made to reference parameters inside procedure bodies update the values hidden behind these actual variable parameters. A procedure may thus be called in different contexts with different variable parameters.[1]

Some imperative languages, e.g., PASCAL, also support the notion of functions that explicitly return values to their calling contexts by assignments to the function identifiers. However, this distinction is somewhat misleading since functions may also have reference parameters that may or may not be used for environment updates, i.e., such functions are in fact procedures as well. In the following, we will therefore uniformly refer to abstractions in the imperative world as procedures, keeping in mind that functions are just special cases of procedures that do not have (or make assignments to) reference parameters and produce explicit return values.[2]

With assigments and procedures that are to perform explicit environment updates, we end up with a degenerate form of the λ-calculus in which variables have lost their status as first-class objects: as before, they may occur as actual parameters (or in operand positions) of procedure calls, but what is substituted for the formal parameters are the values they represent, not the variables themselves, and procedures cannot return variables by assignments to other variables.

For the same reason, we may not have the equivalent of full normalization either. There is no way for partially applied (or unapplied) procedures to perform any useful computations since there would be no valid values for the uninstantiated formal parameters. This has led to defining for all known imperative languages a semantics which demands that procedures be legitimately applied to full sets of operands (or actual parameters) only, though this restriction is unnecessarily severe. There is nothing conceptually that stands in the way of turning partial applications into closures and passing them along as procedures of lesser arities. The decision not to support such closures is largely of a pragmatic nature as it simplifies the implementation on an underlying abstract or real machine, specifically the compilation to executable code and the structure of the runtime environment. Though this in fact rules out the computation of new from existing procedures, there is at least a mecha-

[1] Assignments may also be made directly to nonlocal variables if a procedure is used in just one context.

[2] Interestingly enough, in a predominantly function-based language like SCHEME, all abstractions are called procedures, whereas in an imperative language like C, all abstractions are called functions.

nism available by which procedures may be parameterized by other procedures passed along as operands. This mechanism must inevitably employ something akin to closures since actual procedure parameters must carry with them the environments in which nonlocal variables are instantiated.

The ensuing programming style is called procedural (or imperative) since it is primarily concerned with organizing computations as sequences of state changes effected by procedures and assignments that, loosely speaking, must ensure that the right things (data values) are in the right places (boxes) at the right times (or states of program execution). Thus, programming is not just a matter of specifying what is to be computed, as with an expression-oriented language such as AL, but to a large extent also a matter of detailing how, on a rather granular level, the computation must be performed in terms of interactions with the environment and of the causes and (side)effects that, beyond purely logical dependies, may thus be introduced among individual operations. However, as we will see later in this chapter, an abstract machine that supports this programming model looks surprisingly similar to the functional abstract machine of the preceding chapter, merely requiring a few more instructions to deal with explicit updates to environment entries.

13.1 Outline of an Imperative Kernel Language

Other than using assignment statements and procedures instead of functions, programs of high-level imperative languages such as for instance PASCAL, MODULA or C are basically constructed in the same way as AL programs.

They are composed of sets of procedure declarations and statement blocks. A procedure is an abstraction consisting of a header and a body. The header defines a procedure name (or identifier) and a list of formal parameters that typically includes one or more of the aforementioned reference parameters. The body specifies, either as a single compound statement or as a sequence of statements embedded in a block, the computation to be performed by the procedure or, more precisely, its effect on the environment. In some of the languages, e.g., PASCAL or MODULA, statement blocks may be preceded by definitions of locally used procedures, which permits the unbounded nesting of such definitions. Other languages such as C permit just one level of (or flat) procedure declarations.

The main program is itself a procedure with a unique header and a body.

The scopes of variables declared in the header of a procedure stretch over its entire body. The same applies to local variable and procedure definitions within a body. With respect to these definitions, we have the same scoping rules as in the λ-calculus.

A statement block may include multiple assignments to local variables, to global variables declared in an enclosing procedure and to reference parameters declared in the header. It may also include an assignment of a return

value to the procedure identifier, which is usually the last (or the only) one of the block.

Almost all imperative languages are statically typed, using a monomorphic type system; formal parameter and local variable declarations must include explicit type specifications. Programs are executable only if they are well typed. However, in this chapter we will largely ignore typing and simply assume that all programs under consideration have successfully passed type checking.

The syntax of a simple imperative kernel language IL that we will use in this chapter is given in Fig. 13.1. It has intentionally been chosen to resemble the syntax of AL (see p. 40) rather than the traditional syntax of imperative languages, to expose more clearly some of the similarities of program construction and also of compiling IL programs to abstract machine code.

$$
\begin{aligned}
\mathit{program} &=_s \texttt{main} = \mathit{body} \\
\mathit{body} &=_s \{\ \mathit{statements}\ \} \mid \\
&\quad\ \texttt{defs}\ \mathit{procedures}\ \texttt{in}\ \mathit{body} \mid \\
&\quad\ \texttt{bind}\ \mathit{vars}\ \texttt{in}\ \mathit{body} \\
\mathit{procedures} &=_s \mathit{procedure} \mid \mathit{procedures};\ \mathit{procedure} \\
\mathit{procedure} &=_s \texttt{proc}\ \mathit{proc\_id} = \texttt{sub}\ \mathit{form\_pars}\ \texttt{in}\ \mathit{body} \\
\mathit{statements} &=_s \mathit{statement} \mid \mathit{statements};\ \mathit{statement} \\
\mathit{statement} &=_s \mathit{var} := \mathit{expr} \mid *\mathit{var} := \mathit{expr} \mid \mathit{proc\_call} \mid \\
&\quad\ \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{statements}\ \texttt{else}\ \mathit{statements} \\
\mathit{expr} &=_s \mathit{const} \mid \mathit{var} \mid (\ \mathit{un\_op}\ \mathit{expr}\ ) \mid (\ \mathit{bin\_op}\ \mathit{expr\_1}\ \mathit{expr\_2}\ ) \mid \mathit{proc\_call} \\
\mathit{form\_pars} &=_s \mathit{par} \mid \mathit{form\_pars}\ \mathit{par} \\
\mathit{par} &=_s \mathit{var} \mid *\mathit{var} \\
\mathit{proc\_call} &=_s (\ \mathit{proc\_id}\ \mathit{act\_pars}\ ) \\
\mathit{act\_pars} &=_s \mathit{act\_par} \mid \mathit{act\_pars}\ \mathit{act\_par} \\
\mathit{act\_par} &=_s \&\mathit{var} \mid \mathit{expr}
\end{aligned}
$$

Fig. 13.1. The syntax of a simple imperative kernel language IL

From top to bottom, this syntax breaks down as follows.

A program is a `main` procedure with a body. A body may be a block of statements, a sequence of procedure definitions preceded by the keyword `defs` and followed by a body, or a `bind` construct for the definition of local variables

in a body. The `defs` construct is equivalent to a `letrec` expression of AL in that it may be used to define mutually recursive procedures (of which at least one should be called in its body), the `bind` construct bears some resemblance to the `let` expression of AL in that it assigns values to the variable occurrences bound by it, but it does so from within the body.[3]

A procedure definition is preceded by the keyword `proc`. The left-hand side of the defining equation gives the procedure a name (or identifier) *proc_id*, the construct on the right that is preceded by the keyword `sub` replace the `lambda` abstractions of AL: its binds the formal parameters specified in the sequence *form_pars* in the procedure body. The keyword `sub` denotes an abstractor that signifies naive substitution of the variable occurrences it binds by actual parameters. If the variables happen to be reference parameters, they may also be used to effect updates in the environment.

The statements which do the actual computing include assignments of expression values to variables, procedure calls, and `if_then_else` clauses whose consequents and alternatives are again sequences of statements. Expressions may be constants, variables, applications of unary and binary operators, and procedure calls that return values.

This is basically what we need to know about the construction of IL programs to design an abstract machine that executes them. What follows next in Fig. 13.1 are just the nitty-gritties of parameter declarations and of the actual parameters for procedure calls.

Formal procedure parameters *form_pars* may be value parameters, denoted as ordinary variables *var* or reference parameters denoted as $*var$. Procedure calls have the identifiers *proc_id* applied to sequences of actual parameters. These parameters are either expressions substituting for value parameters or pointer variables, denoted as $\&var$, which substitute for reference parameters.

As stated at the beginning of this chapter, the semantics of imperative languages does not support partial applications. Procedures must be applied to exactly as many operands (of compatible types) as there are formal parameters. However, most imperative languages permit procedures to be passed as procedure parameters, with a similar effect: a calling procedure may be specialized to perform different computations depending on the procedures by which these parameters are instantiated. A procedure passed as parameter to another procedure, of course, must carry along the environment for the instantiation of its nonlocal variables in order to execute correctly in the calling procedure's body. However, as the mechanism to this effect is basically the same as in Sect. 12.1.2, we can forgo looking at procedure parameters since it adds nothing new and keeps the description of the abstract machine that follows in Sect. 13.3 focussed on the essentials.

Like almost all imperative languages, IL is assumed to have a call-by-value semantics (or an applicative order semantics in λ-calculus terminology). Calling

[3] Multiple assignments within a statement block to the same variable may be equivalently expressed in purely functional form as nestings of `let` expressions.

a parameter by value means that an operand value must be substituted directly, calling it by reference means that the procedure must receive a pointer (a reference) to an operand value. Calling a parameter by reference may imply sharing a single value between several pointer occurrences.

13.2 An Example of an IL Program

For the sake of a simple but nontrivial example program we add to our kernel language arrays of basic values of the same type, denoted as

$$vec =_s < a\_1 \ldots a\_i \ldots a\_n > \ ,$$

and selector operations $vec[\ i\ ]$ that, when occurring on the left-hand side of an assignment statement, update the i-th array entry with the value of the expression on the right-hand side and, when occurring in an expression, represent the value of the i-th array entry a_i.

Arrays are composite objects and thus prime candidates for being passed to procedures as reference parameters for in-place updating.[4]

Perfect examples of programs that use reference parameters derive from algorithms that sort the elements of an unsorted array of numbers, say, in monotonically ascending order. The least sophisticated of these algorithms is bubble-sort which works like this: the array is traversed from left to right (or in the direction of increasing index i) and numbers are swapped if the one in position i happens to be greater than the one in position $i + 1$. In the first pass through the array the greatest number thus moves step by step to the right (or into the highest index position n) but the numbers in all other positions remain unsorted. Another such pass through the array that ignores the last element (or the largest number) brings the second largest number to the right, and so on. Thus, if the array contains n numbers, it takes $n - 1$ passes to sort the entire array, with the i-th pass extending over the leftmost $n + 1 - i$ numbers only.

Figure 13.2 shows how this algorithm may be implemented as an IL program. It uses three procedures nested inside each other. The outermost `defs` construct defines a tail-recursive procedure *sort* that merely controls the $n-1$ sorting passes through an n-ary array. This procedure receives as reference parameter the array to be sorted and as value parameter the length of the array. It recursively calls the procedure *bubble*, defined by another `defs` construct local to *sort*, that does the actual sorting. The array is passed on to *bubble* again as a reference parameter; the parameter i specifies the actual index po-

[4] For performance reasons it is advisable to pass all composite objects by reference, even if they are not modified by the procedure. The difference in runtime complexity becomes evident when considering an $n * m$ matrix: it takes one unit of time to pass it by reference but $n * m$ units of time to pass it by value.

sition. Depending on the outcome of the comparison, *bubble* calls yet another procedure *swap*, defined local to it, that simply flips the two elements.[5]

```
main = bind lvec in
          defs
            proc sort =
              sub n *vec in
                bind i in
                  defs
                    proc bubble =
                      sub i *vec in
                        bind j in
                          defs
                            proc swap =
                              sub i j in
                                bind temp in {
                                    temp := *vec[ i ];
                                    *vec[ i ] := *vec[ j ];
                                    *vec[ j ] := temp
                                    }
                          in { if ( gt i n )
                                  then j := ( + 1 i );
                                       if ( lt *vec[ i ] *vec[ j ] )
                                       then ( swap i j )
                                       else nop ;
                                       ( bubble j *vec )
                                  else ( sort ( - 1 n ) *vec )
                               }
                    in { if ( gt 1 n )
                            then i := 1;
                                 ( bubble i *vec )
                            else nop
                       }
          in { lvec := < 5 4 9 8 1 6 3 2 7 >;
               ( sort &lvec 10 )
                    ...
             }
```

Fig. 13.2. An IL implementation of the bubble-sort algorithm

The *main* program is a `bind` construct for the variable *lvec*, with the nested `defs` constructs as its body. The statement block of the outermost `defs` makes an assignment to *lvec* of the array that is to be sorted and then applies the procedure *sort* to the pointer &*lvec* and to the actual number of elements in the array, which is 10 in this particular case.

[5] Note that the alternatives of two of the `if_then_else` clauses use the primitive `nop` for no operation.

The important point here is that, throughout the entire program run, there exists only this one copy of the array to which *&lvec* points. This pointer, not the array itself, is replicated when passing it from the procedure *sort* to *bubble* and further on to *swap*. That is to say, the array can be sorted completely in place, i.e., without ever producing a new (updated) copy of it, thus saving both memory space and execution time.[6] Assuming that the elements of the array are held in consecutive locations of the addressable memory, individual array elements $vec[\ i\ ] \mid 1 \leq i \leq n$ may be accessed simply by adding the index i to this pointer.

However, in-place updating cannot be had if arrays, or other composite objects for that matter, with different element values have to coexist in the course of executing a program. But it is then the programmer's responsibility to explicitly create and manage these different versions of the same objects, as opposed to AL, where this is none of her/his concerns.

13.3 The Runtime Environment

A good starting point for the design of an imperative abstract machine, in the following abbreviated as IAM, is the SECD_I machine of the preceding chapter. Both machines share support for a call-by-value semantics that, in λ-calculus terminology, is only weakly normalizing, and abstractions can only be applied to full sets of actual parameters (or arguments).

We remember that the SECD_I machine features a code structure C traversed by a pointer pc, an environment structure E with a top pointer pe, a value stack S, a dump D for return continuations of function calls, and a heap H. They are the structures that need to be included in the IAM as well, the question is how this should (or could) be done most appropriately.

Looking first at the environment E, we are faced in the case of the SECD_I machine with the necessity of realizing it as a cactus structure of argument frames for function calls. This is a consequence of wrapping partially applied or unapplied functions in closures that carry along with them the associated environments. As these closures may hold on to their environments beyond the function calls that created them, there is no way of releasing them until the closures have also ceased to exist.

The situation is quite different with the IAM. Here we have by definition no partial applications and hence no closures to support. There are only the special cases of procedure parameters that need to carry along the environments for their nonlocal variables to be taken care of. Now, since these environments are either the same or earlier environments of the calling procedures and the actual procedure parameters are applied in the calling procedure's statement

[6] Some print procedure following the call of *sort* in the outermost statement block and applied to *&lvec* (which is not shown) would print the sorted array on an output medium.

blocks, we can be sure that no environment frame needs be kept beyond the lifetime of the procedure call that brought it about. Environment frames may therefore be released in reverse order of creation, following a last-in-first-out (or LIFO) discipline, with the pleasant consequence that the environment can be realized as a stack in which frames can be packed densely on top of each other.

This would leave us with three stacks E, D and S that are getting involved in procedure calls. Following SECD_I machine ritual, the calling procedure would first build an argument frame in S, entering the called procedure would then push the tuple (pc, pe) of the current program counter | environment pointer pair onto the dump stack D, and move the argument frame from S to what has now become a stack E, whereupon the called procedure would use S as a working stack for local variable instantiations and temporaries. However, since these items could be removed in reverse order upon returning from a procedure call, the three stacks may, as a matter of economy, be merged into a single runtime stack, just as in the G-machine of Chap. 9.

With this in mind, we can now define the IAM as comprising

- a code structure C that holds, as a sequence of IAM instructions, the entire program code;
- a program counter (pointer) pc that identifies the instruction currently executed from C;
- a runtime stack S that holds what we will refer to as activation records of procedure calls, of which each is composed of an argument frame, a return continuation and a workspace for local variables and temporaries;
- a stack pointer ps pointing to the current top of stack S;
- a pointer pe to the environment deeper down in stack S of the procedure call that is currently active;
- a heap H for nonatomic values such as arrays, records etc.

That is, a state of the IAM is described by a six-tuple (pc, ps, pe, S, C, H).

As in the SECD_I machine, the code and the stack are held in nonoverlapping sections of addressable memory; all instructions and stack entries are assumed to have unit length. Stack entries are either basic values or pointers deeper down into the stack or into the heap. Following standard convention, the stack is assumed to grow toward lower addresses, and the stack pointer ps points to the next (empty) entry to be filled by a push operation.

An important issue of working with just a single runtime stack concerns the problems of how we must

- arrange the various entries within an activation record so that they can be conveniently accessed, using fixed offsets relative to some reference position within the record;
- dynamically link the activation records in their order of creation;
- statically link the activation records in compliance with the nestings of procedure definitions in the source program in order to be able to access instantiations of nonlocal variables.

Because of the sequence of actions that bring them about, there is not much of a choice as to where we have to place argument and workspace frames relative to each other, but there is some degree of freedom in deciding how activation records need to be linked up to each other, where to put the link pointers, and relative to which point of reference we wish to address record entries.

In the following, we will discuss two possible solutions to this problem that basically differ with respect to the linking of activation records and, in consequence, to their layout.

13.3.1 Using Static and Dynamic Links

The first solution uses both static and dynamic links to connect the activation record of a procedure call to its own environment and explicitly to the environment of the calling procedure, respectively. The dynamic link, together with the program counter value at the point of a procedure call, in fact constitutes the return continuation of that call.

Figure 13.3 shows conceivable stack configurations with statically and dynamically linked activation records before and after a procedure f calls another procedure g. Both f and g are assumed to be locally defined on the same level under another procedure h. On the left, we have just the activation record of f on top of the activation record of h. At the lower end of the record of f (which is at higher addresses) we have an argument frame, and on top of it, a dynamic link $dlink\_f$ followed by a static link $slink\_f$, both of which in this particular case point to the activation record of h and, more specifically, to the static link entry $slink\_h$ therein. The current environment pointer pe points to the $slink\_f$ entry.

The next entry is the program counter value pc_f that is pushed when entering the code of f. This value, together with the dynamic link $dlink\_f$ that is stacked up underneath, completes the return continuation of f.

Finally, we have in the stack a workspace frame for local variables and for temporaries.

On the right of the figure, we see how the stack grows when f calls the procedure g. First the procedure f creates the argument frame for g, and then pushes both the dynamic and the static link pointers $dlink\_g$ and $slink\_g$ on top of it. Since g is defined at the same level as f, the static link $slink\_g$ happens to be the same as $slink\_f$; otherwise, it could be obtained by dereferencing $slink\_f$ and thus point deeper down into the stack.

The pointer new_pe to the $slink\_g$ entry overwrites the current environment pointer pe. Branching to the procedure g saves the current program counter pc_g to complete its return continuation (the other part being $dlink\_g$).

Now the code of the procedure g pushes the part of the workspace that holds the local variables, simply by advancing the stack pointer to the position

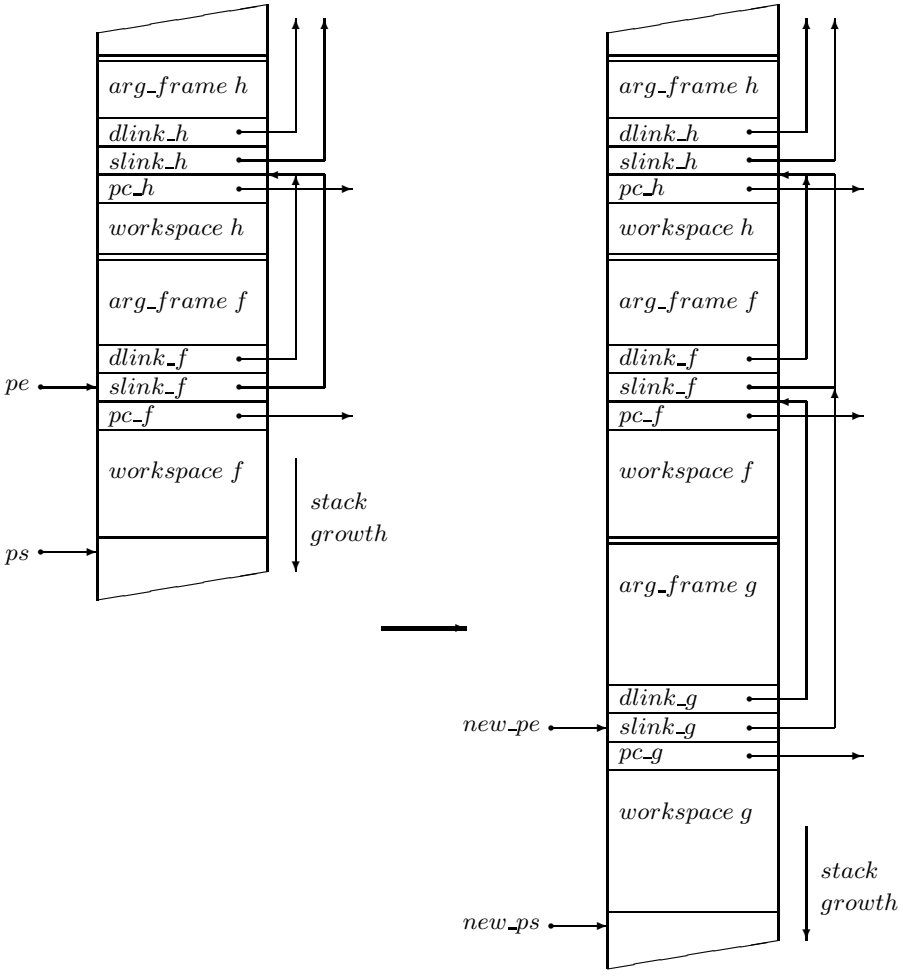


Fig. 13.3. Typical stack configurations before and after a procedure f calling a procedure g

new_ps, and then starts computing its statement block, which may further expand (and subsequently shrink) the workspace.

Before returning from g, the workspace is cleared by setting the stack pointer *ps* back by the number of entries that are still left there (which are the local variables). The pointer that then pops to the top is taken as the return address of the calling procedure f, and the second entry underneath is used to restore the pointer *pe* that renders the static link *slink_f* to the environment of f accessible. Finally, having returned from g, the procedure

f, knowing how many entries it has pushed before calling g, releases the rest of the activation record of g by resetting the stack pointer ps accordingly.

Though this solution has the pleasant property of having both link pointers and the program counter stacked up in adjacent locations of a control block[7] that is squeezed between the argument frame and the workspace, this is not the smartest thing to do. In the next subsection, we will show that, with a more appropriate placement of the static link pointer, we can completely do without the dynamic links and also facilitate addressing within the records.

13.3.2 Dropping Dynamic Links

The trick that needs to be played to eliminate dynamic links from activation records is to place the static link pointers at the very bottom, right on top of the workspaces of the calling procedures, to freeze the environment pointers to these stack positions, and to access record entries relative to them. This can be done consistently throughout all activation records that belong to the environment of a particular procedure call.

Consider again the situation in which a procedure f calls another procedure g, as depicted in Fig. 13.4. On the left, we have in the stack the activation record of f sitting on top of the activation record of the procedure h, with the environment pointer pe pointing to its bottom, where the static link $slink\_f$ to the environment of f, i.e., to the corresponding entry at the bottom of the activation record of h, can be found. Between the argument and workspace frames is saved just the return address of f, and the stack pointer ps again points to the first empty entry on top of the workspace. This is exactly the position where we have to place the static link pointer for the procedure g. The first step of doing this is to reserve an empty slot for this pointer. Next, the procedure f creates the argument frame for g (which may require doing evaluations in the environment referenced by pe), then accesses with the pointer pe the link $slink\_f$ to get the static link $slink\_g$ to the environment of g, writes this link into the stack position reserved for it, and finally sets the new environment pointer new_pe to this position. Calling the procedure g pushes just the return address pc_g, whereupon g pushes an as yet uninstantiated workspace frame for its locals and advances the stack pointer ps accordingly, resulting in the stack configuration shown on the right of the figure.

Accessing locals and value parameters of g can be done with fixed negative offsets figured out by the compiler relative to the static link pointer new_pe. Accesses to nonlocal variables and procedure parameters may be accomplished by dereferencing the static links beginning at new_pe and, again, by addressing relative to the address thus found, which should point to the bottom of the desired activation record.

Returning from the procedure g involves g itself to clean up the workspace, which pops the return address pc_g to the top of the stack. Then the calling

[7] In an implementation on a real computing machine this control block would include a few more entries.

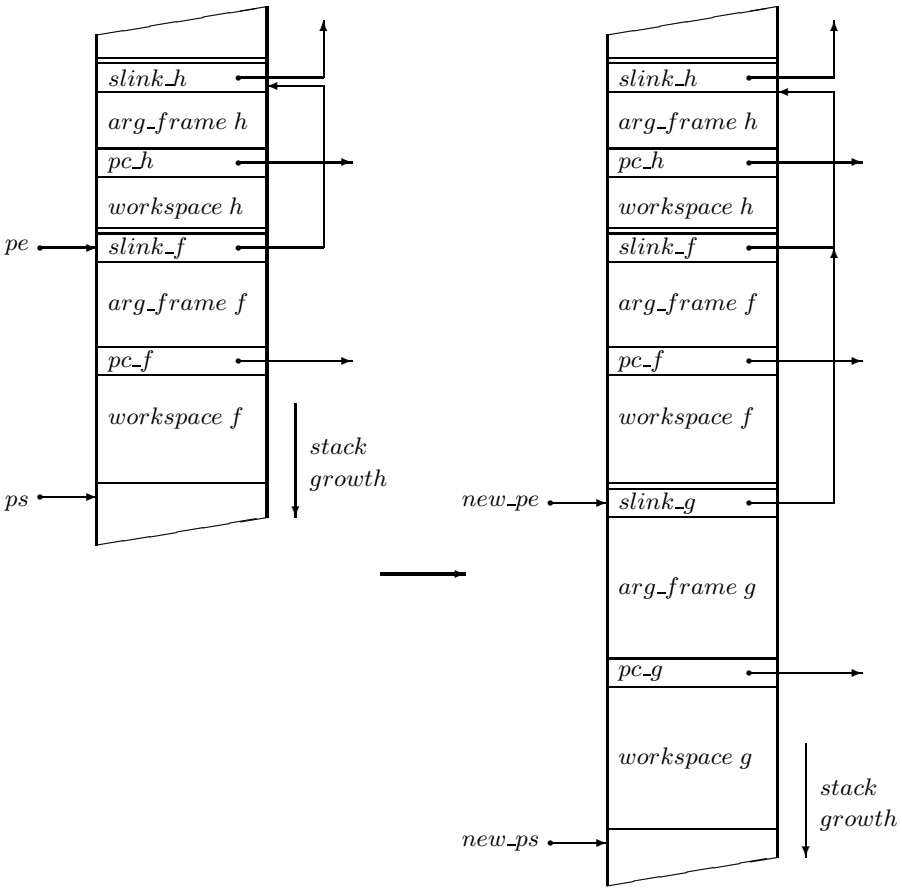


Fig. 13.4. An improved stack configuration before and after a procedure f calling a procedure g

procedure f, upon returning to it, cleans up the rest of the stack down to *slink_g*. Adding the (known) size of the activation record of f to this stack position gives the environment pointer *pe* of the procedure f and thus restores the stack configuration on the left of the figure, except that a few entries may have been updated.

If a procedure behaves like a function in that it returns a value rather than just changing its environment, another entry must be reserved on the stack through which this value may be handed over to the calling procedure. The place where this entry belongs is on top of the workspace of the calling procedure and underneath the static link pointer of the called procedure,

where it can be conveniently accessed from both sides which, incidently, closely resembles the result-passing mechanism of the G-machine (see Sect. 9.5).

13.3.3 Calculating Stack Addresses ∗

Most of what needs to be known about translating variable occurrences into stack accesses (or addresses) has been explained in Sect. 12.1.2. There we introduced a tuple notation that pairs occurrences of function identifiers with indices specifying binding distances, in terms of intervening `lambda` binders, to the `lambda` that binds the function parameters. An equivalent index tuple notation was introduced for occurrences of `lambda`-bound variables, of which the first index denotes the distance to the binding `lambda`, again in numbers of intervening `lambda`s, and the second index gives the declaration position within the list of variables bound by it.

This tuple notation can be used in the IAM more or less as it is and directly translated into dereferencing static links and into offsets relative to the bases of activation records, assuming that their layout is as described in the preceding subsection.

To convey the basic idea, we consider a fragmental procedure definition

```
proc ff = sub v1 ... * vi ... vn in
     /*local procedure | function definitions*/
     bind u1 ... uj ... um in {
                ...
           ( ff ... &uj ... );
                ...
           uj := ... ;
                ...
          * vi := ... uk ... vl ... ;
                ...
     } .
```

This procedure features n formal parameters $v_1 \ldots v_n$, of which $*v_i$ is assumed to be a reference parameter, and m local variables $u_1 \ldots u_m$, all of them being bound in the statement block enclosed in the curly braces { }. Thus, an activation record for ff must include an argument frame of n entries and a workspace of m entries, on top of which may follow some temporaries.

Taking the static link pointer entry as the base of an activation record, the argument frame then obviously occupies offset positions within the interval $[-1, \ldots, -n]$ and, accounting with one slot for the program counter entry pc, the static part of the workspace can be found in offset positions that are within the interval $[-(n+2), \ldots, -(n+m+1)]$ relative to this base.

To compute bases and offsets of stack entries, a preprocessor needs to replace the `sub`s and `bind`s of this procedure with nameless abstractors Δ_s and

Δ_b, respectively, and the variable occurrences bound by them need to be accordingly replaced with index tuples $[\ i\ j\ ]_s$ and $[\ i\ j\ ]_b$. We thus obtain

$$
\begin{array}{l}
\texttt{proc}\ ff = \underbrace{\Delta_s \ldots * \Delta_s \ldots \Delta_s}_{n}\ \texttt{in} \\
\quad /*\text{local procedure} \mid \text{function definitions}*/ \\
\quad \underbrace{\Delta_b \ldots \Delta_b \ldots \Delta_b}_{m}\ \texttt{in}\ \{ \\
\qquad \ldots \\
\qquad (\ [\ 1\ ff\ ] \ldots \&[\ 0\ (m-j)\ ]_b \ldots\ ); \\
\qquad \ldots \\
\qquad [\ 0\ 0\ ]_b := \ldots; \\
\qquad \ldots \\
\qquad * [\ 0\ (n-i)\ ]_s := \ldots [\ 0\ (m-k)\ ]_b \ldots [\ 0\ (n-l)\ ]_s \ldots \\
\qquad\quad \ldots \\
\quad \}\ .
\end{array}
$$

Using Δ-abstractors instead of Λs signifies a departure from the nameless Λ-calculus in that they not only bind parameters to be substituted by (the values of) expressions but also reference parameters and box variables that are subject to assignments, and thus have another quality.

We note that the index tuples that have replaced `sub-` or `bind`-bound variable occurrences have distance indices 0, i.e., they are bound by the respective innermost sequences of Δ-abstractors, and the second indices give the declaration positions within those sequences. For instance, the index tuple $[\ 0\ (m-j)\ ]_b$ that replaces the variable occurrence u_j is bound by the $(m-j)$-th binder from the right of the innermost sequence $\underbrace{\Delta_b \ldots \Delta_b}_{m}$; likewise, the index tuple $[\ 0\ (n-i)\ ]_s$ that replaces the variable occurrence v_i is bound by the $(n-i)$-th binder from the right of the innermost sequence $\underbrace{\Delta_s \ldots \Delta_s}_{n}$. The distance index of 0 in both cases means that the entries can be found in the topmost activation record on the stack, and the base address that must be used is the current environment pointer *pe* that points to the link entry of this record.

The occurrence of the procedure identifier ff in the statement block is replaced by the tuple $[\ 1\ ff\ ]$, meaning that the environment of ff must be obtained by dereferencing the pointer *pe* once (or, more directly, that the link entry of the topmost activation record must be copied).

As outlined in Sect. 12.1.2, occurrences of tuples with distance indices $i > 1$ must have the actual environment pointer *pe* dereferenced i times to get to the base of the activation record relative to which an entry must be accessed.

We now have to figure out how the declaration positions j of the index tuples $[\ i\ j\ ]_x$ need to be translated into offsets relative to the bases of the

activation records. Beginning with the Δ_s-bound index tuples, we note that a legitimate application of a procedure

$$ff = \texttt{sub}\ v_1 \ldots\ v_n\ \texttt{in} \ldots \{\ldots v_{(n-j)} \ldots\}$$

to n actual parameters $ppar_1, \ldots, ppar_n$ has the preprocessed form

$$([\ i\ ff\ ]\ ppar_1 \ldots ppar_n) =_s \\ ([\ i\ \underbrace{\Delta_s \ldots \Delta_s}_{n}\ \texttt{in} \ldots \{\ldots [\ 0\ j\ ]_s \ldots\}\ ]\ ppar_1 \ldots ppar_n)\ .$$

Using λ-calculus conventions, we know that the arguments $ppar_{(n-j)} \mid\ j \in \{1, \ldots,\ n\}$ are substituted for occurrences of the index tuples $[\ 0\ j\ ]_s$. This suggests pushing the arguments onto the stack in the order from right to left, in which case they can be accessed relative to the base pe of the argument frame with negative declaration position indices $-j$ as offsets. Distance indices $i > 0$ just mean that the environment pointer pe must be dereferenced i times to get to the right frame base.

Likewise, when preprocessing a block of the form

$$\texttt{bind}\ u_1 \ldots u_m\ \texttt{in}\ \{\ldots;\ u_{m-j} := \ldots; \ldots\}\ ,$$

we get

$$\Delta_b \ldots \Delta_b\ \texttt{in}\ \{\ldots; [\ 0\ j\ ]_b := \ldots; \ldots\}\ .$$

Here we are free to choose the same ordering of indices as above for entries relative to the base of the workspace, i.e., relative to the base pe of the entire activation record we have offsets $-(n+2+j) \mid j \in \{\ 0, \ldots, m-1\ \}$. Thus, translating occurrences of index tuples of either kind into stack addresses becomes a straightforward matter, as in the SECD_I machine described in the preceding chapter, the only difference being that the frames are turned upside down with respect to addressing.

Things are a bit trickier with occurrences of array selectors $*vec[\ h\ ]$. Let $*vec$ denote the dereferencing of a pointer to an array, then the address of the h-th entry in this array may be obtained by adding to this pointer the current index value h. Since h itself is either a `sub`- or a `bind`-bound variable and thus translates into an index tuple as well, we may simply use the notation $[\ i_{vec}\ j_{vec}\ ]_s \mid [\ i_h\ j_h\ ]_{s|b}$ to represent $*vec[\ h\ ]$ in preprocessed form. This notation must be interpreted as using first $[\ i_{vec}\ j_{vec}\ ]_s$ and then $[\ i_h\ j_h\ ]_{s|b}$ to retrieve first the array pointer and then the index value for h, respectively, from deeper down in the stack, adding them up, and using the (heap) address thus obtained to access the selected array element.

13.4 The Instruction Set

We are now ready to put together the IAM, based on the stack representation of the runtime environment as described in Sect. 13.3.2 and on accessing this environment as outlined in the preceding subsection.

As usual, the IAM instructions are defined by a state transition function

$$\tau_{iam} : (\ pc,\ ps,\ pe,\ S,\ C,\ H\ ) \ \rightarrow\ (\ pc',\ ps',\ pe',\ S',\ C,\ H'\ )\ .$$

Since no partial applications need to be dealt with and all the actions are taking place on one runtime stack, the IAM needs fewer and simpler instructions than the SECD_I machine to handle procedure calls. However, a single instruction must be added to update, or assign values to, environment entries.

The IAM instructions that are shared more or less one-to-one with the SECD_I machine are the following:

PUSH *atom* pushes an atomic value or a pointer to a composite value such as an array onto the stack.

COPY $i\ j$ copies to the top of the stack the entry found at an offset $-j$ relative to the base of the activation record obtained by dereferencing i times the current environment pointer pe.

BRC $p_t\ p_f$ realizes a conditional two-way branch: depending on the Boolean value found on top of the stack, code execution continues at either the pointer p_t or the pointer p_f, and the current program counter pc is saved as the address to which control must return after either piece of code has been executed.

BSR p_{ff} realizes a procedure call: it branches unconditionally to the code beginning at the pointer p_{ff} and saves on the stack the program counter pc as the return continuation.

RET returns from a procedure call or from a conditional branch, taking the return address off the top of the stack.

LINK $h\ k$ dereferences h times the current environment pointer pe, writes the pointer thus obtained into the entry found at an offset k relative to the current stack top position ps, and then updates the environment pointer to point to this new position, i.e., $pe := ps + k$.

ALLOC n allocates space for n entries in the stack by moving the stack pointer ps by this number of positions ahead toward lower addresses.

FREE n deallocates the topmost n stack entries by moving the stack pointer ps by this number of positions back toward higher addresses.

The instruction that performs assignments to environment entries is

ASSIGN $i\ j$ which overwrites (or updates) with a value taken from the stack top a stack entry found by dereferencing i times the environment pointer pe and adding to the pointer thus obtained an offset $-j-1$.

Of course, the full instruction set of the IAM also includes the usual variety of arithmetic | logic | relational and array-processing instructions. These instructions take (pointers to) their arguments off the stack top and push (pointers to) result values instead.

Indexed accesses to arrays are handled by another two instructions, of which

FETCH takes the topmost stack entry as a heap address and replaces it with the value found at this address.

STORE again takes the topmost stack entry as a heap address, updates the value at this address by the second stack entry from the top, and then pops both entries.

These instructions allow us to compile, as an example, the bubble-sort program of Fig. 13.2 to IAM code.

We also have the instructions ENTRY and EXIT to mark entry into and exit from code execution.

13.5 Compiling IL Programs to IAM Code *

Defining a compilation scheme for IL is not as straightforward as for an AL kernel. Not only do we have to deal with a slightly more complex syntax, there are also some semantic restrictions that need to be obeyed. Procedures may be legitimately applied to full sets of arguments (or actual parameters) only, and procedures, pointers and values may be passed to procedures only through formal parameters expilicitly so designated. This is essentially a matter of appropriate typing according to the rules of some underlying type system, which we have earlier decided to simply take for granted. Only with this in mind may we take the liberty of again defining compilation simply as a mapping of syntactical constructs (or figures) into pieces of IAM code. Otherwise, we would have to explicitly include typing in the language and type checking in the compilation process to make sure that the IAM code is well typed and thus working as intended.

The IL syntax given in Fig. 13.1 suggests that compilation works with two schemes $\mathcal{B}$ and $\mathcal{C}$, of which the former applies to bodies, procedures and (sequences of) statements, and the latter applies to expressions. Both schemes realize mappings of the general form

$$\mathcal{Z}[\, s\_form : rest \mid n \,] \implies code[\, s\_form \,];\ \mathcal{Z}[\, rest \mid n \,] \ ,$$

where s_form denotes a legitimate syntactical form, $rest$ denotes some remaining sequence of the same kind (which may be empty), and the parameter n specifies the number of parameters passed to the code to be generated. An irrelevant parameter n is denoted by '-'.

The top-level compilation schemes $\mathcal{B}$ is defined in Fig. 13.5. It partitions into two sets of rules, of which those that belong to the first set basically apply to procedure definitions. More specifically, when B is applied to

- the main program, it is driven in front of its body, embedded in ENTRY and EXIT instructions; the ensuing code is referenced by a pointer *pp* (rule (1)).
- a `defs` construct, it first generates the code for its body, followed by the codes for the procedures defined under it (rule (2)).

(1) $\mathcal{B}[\,\texttt{main} = body \mid -\,] \implies pp \rightsquigarrow \text{ENTRY};\ \mathcal{B}[\,body \mid -\,];\ \text{EXIT};$

(2) $\mathcal{B}[\,\texttt{defs}\ proc\_1 \ldots proc\_k\ \texttt{in}\ body \mid n\,]$
$\implies \mathcal{B}[\,body \mid n\,]; \mathcal{B}[\,proc\_1 \mid -\,]; \ldots;\ \mathcal{B}[\,proc\_k \mid -\,]$

(3) $\mathcal{B}[\,\underbrace{\Delta_b \ldots \Delta_b}_{m}\ \texttt{in}\ body \mid n\,] \implies \text{ALLOC}\ m;\ \mathcal{B}[\,body \mid n\,];\ \text{FREE}\ m$

(4) $\mathcal{B}[\,\texttt{proc}\ proc\_id = \underbrace{\Delta_s \ldots \Delta_s}_{n}\ \texttt{in}\ body \mid -\,] \implies p\_proc\_id \rightsquigarrow \mathcal{B}[\,body \mid n\,];\ \text{RET}$

(5) $\mathcal{B}[\,\{\ statements\ \} \mid n\,] \implies \mathcal{B}[\,statements \mid n\,]$

(6) $\mathcal{B}[\,[\ i\ j\ ]_b := expr;\ statements \mid n\,]$
$\implies \mathcal{C}[\,expr \mid n\,];\ \text{ASSIGN}\ i\ (n + j + 2);\ \mathcal{B}[\,statements \mid n\,]$

(7) $\mathcal{B}[\,*[\ i\ j\ ]_s := expr;\ statements \mid n\,]$
$\implies \mathcal{C}[\,expr \mid n\,];\ \text{ASSIGN}\ i\ (j + 1);\ \mathcal{B}[\,statements \mid n\,]$

(8) $\mathcal{B}[\,(\ [\ h\ proc\_id\ ]\ act\_par_1 \ldots act\_par\_r\ );\ statements \mid n\,]$
$\implies \text{ALLOC}\ 1;\ \mathcal{C}[\,act\_par\_r\,];\ \mathcal{C}[\,act\_par\_r - 1\ \ldots act\_par\_1$
$[\ h\ prod\_id \mid r\ ] \mid n\,];\ \mathcal{B}[\,statements \mid n\,]$

(9) $\mathcal{B}[\,\texttt{if}\ expr\ \texttt{then}\ \{\ statements\_1\ \}\ \texttt{else}\ \{\ statements\_2\ \};\ statements \mid n\,]$
$\implies \mathcal{C}[\,expr \mid n\,];\ \text{BRC}\ p_t\ p_f;\ \mathcal{B}[\,statements \mid n\,]$
$p_t \rightsquigarrow \mathcal{B}[\,statements\_1 \mid n\,];\ \text{RET};\ p_f \rightsquigarrow \mathcal{B}[\,statements\_2 \mid n\,];\ \text{RET};$

Fig. 13.5. The top-level compilation scheme $\mathcal{B}$ of IL

- a `bind` constructs, it first allocates stack space for the local variables, then generates the code for its body using $\mathcal{B}$, and afterwards releases the stack space again (rule (3)).
- a procedure definition, it applies $\mathcal{B}$ to its body and instantiates the parameter n with its arity (which by definition equals the number of actual parameters that the code can expect to find on the stack). The code is referenced by a symbolic pointer (or label) derived from the procedure identifier (rules (4)).

The second set of $\mathcal{B}$-rules applies to (sequences of) statements, recursively calling on the scheme $\mathcal{C}$ to compile expressions. In particular, $\mathcal{B}$ compiles

- an assignment statement to code generated by $\mathcal{C}$ for the right-hand side expression, followed by the instruction ASSIGN whose index parameter depends on whether the assignment is made to locally bound variables or to reference parameters (rules (6) and (7)).
- a procedure call to code that first allocates in the stack an empty slot as a placeholder for a link pointer, then evaluates the actual parameters in reverse order, and finally branches to the procedure code itself (rule (8)).

- a conditional in that it first generates the code for the predicate expression using $\mathcal{C}$, followed by a conditional branch instruction, and then compiles the consequent and alternative statements by application of $\mathcal{B}$, both followed by return instructions (rule (9)).

In all three cases, $\mathcal{B}$ is recursively applied to the remaining sequence of statements until this sequence becomes empty.

Figure 13.6 lists the compilation rules for expressions that may occur on the right-hand sides of assignment statements, in predicate positions of conditionals, and in parameter positions of procedure calls. The scheme $\mathcal{C}$ applies to constants (including pointers) that have to be pushed onto the stack (rule (10)), to index tuples that have to have the contents of the respective stack entries copied to the stack top (rules (11) and (12)), and to applications of primitive operators that have their argument expressions compiled in reverse order to code that pushes their values onto the stack and then applies the respective instructions to them (rules (13) and (14)).

Applications of procedures that are expressions are assumed to produce return values, i.e., they are in fact function applications. They are by $\mathcal{C}$ compiled to codes for the argument expressions, followed by the code for the procedure call itself (rules (15) and (16)). Rule (15) puts an ALLOC 2 instruction in front of this code to allocate two empty slots on the stack, of which one is reserved for the return value of the procedure call, and the other is to accommodate the link pointer to the calling environment. It also attaches the number k of arguments as another component to the procedure tuple $[\ h\ \mathit{proc\_id}\ ]$ that is used by rule (16) to generate the code that establishes the link to the calling environment, branches to the procedure code and, upon returning, releases the argument frame of size k from the stack. Rule (16) also applies to the compilation of ordinary procedure calls that are treated as statements (see rule (8) of Fig. 13.5).

The special cases of selecting elements from arrays are taken care of by two more rules,[8] of which

$$\begin{aligned}(7a)\ \mathcal{B}[\ *[\ i\ j\ ]_s \mid [\ k\ l\ ]_{s|b} := \mathit{expr};\ \mathit{statements} \mid n\ ] \\ \Longrightarrow\ \mathcal{C}[\ \mathit{expr}\ ];\ \text{COPY}\ i\ j+1;\ \text{COPY}\ k\ (l+n+2);\ \text{ADD};\ \text{STORE}; \\ \mathcal{B}[\ \mathit{statements} \mid n\ ]\end{aligned}$$

handles assignments to individual elements of arrays through reference parameters, and

$$\begin{aligned}(11a)\ \mathcal{C}[\ [\ *[\ i\ j\ ]_s \mid [\ k\ l\ ]_{s|b} : es \mid n\ ] \\ \Longrightarrow\ \text{COPY}\ i\ j+1;\ \text{COPY}\ k\ (l+n+2);\ \text{ADD};\ \text{FETCH};\ \mathcal{C}[\ es \mid n\ ]\end{aligned}$$

[8] The enumeration of these rules refers to the fact that they relate to rules (7) and (11) in that they too make assignments to and retrieve values from reference parameters, respectively.

(10) $\mathcal{C}[\,const : es \mid n\,] \implies \text{PUSH } const;\ \mathcal{C}[\,es \mid n\,]$

(11) $\mathcal{C}[\,*[\,i\ j\,]_s : es \mid n\,] \implies \text{COPY } i\ (j+1);\ \mathcal{C}[\,es \mid n\,]$

(12) $\mathcal{C}[\,[\,i\ j\,]_b : es \mid n\,] \implies \text{COPY } i\ (n+j+2);\ \mathcal{C}[\,es \mid n\,]$

(13) $\mathcal{C}[\,(\,prim\_op\ expr\_1 \ldots expr\_k\,) : es \mid n\,]$
$\implies \mathcal{C}[\,expr\_k \mid n\,];\ \mathcal{C}[\,expr\_(k-1) : \ldots : expr\_1 : prim\_op : es \mid n\,]$

(14) $\mathcal{C}[\,prim\_op : es \mid n\,] \implies \text{PRIM\_OP};\ \mathcal{C}[\,es \mid n\,]$

(15) $\mathcal{C}[\,(\,[\,h\ proc\_id\,]\ expr\_1 \ldots expr\_k\,) : es \mid n\,]$
$\implies \text{ALLOC } 2;\ \mathcal{C}[\,expr\_k \mid n\,];$
$\mathcal{C}[\,expr\_(k-1) : \ldots : expr\_1 : [\,h\ proc\_id \mid k\,] : es \mid n\,]$

(16) $\mathcal{C}[\,[\,h\ p\_id \mid k\,] : es \mid n\,] \mid \implies \text{LINK } h\ k;\ \text{BSR } p\_id;\ \text{FREE } k;\ \mathcal{C}[\,es \mid n\,]$

Fig. 13.6. The compilation rules for expressions

handles occurrences of individual array elements in expressions, e.g., on the right-hand sides of assignments.

Both rules have in common the instruction sequence

$$\ldots;\ \text{COPY } i\ (j+1);\ \ \text{COPY } k\ (l+n+2);\ \text{ADD};\ldots\ ,$$

which computes heap addresses from stack entries given by index quadruples $*[\,i\ j\,]_s \mid [\,k\ l\,]_{s|b}$. The instruction STORE in rule (7a) makes an assignment to this heap address, the instruction FETCH in rule (11a) pushes the value stored at this address onto the stack.

13.6 Compiling the Bubble-Sort Program

We can now study how the compiler works by applying it to the bubble-sort program of Fig. 13.2. Compilation must of course be preceded by a preprocessing step that turns `sub`- and `bind`-bound variable occurrences into index tuples. The preprocessed program text is shown in Fig. 13.7.

Compilation starts out by application of the scheme $\mathcal{B}$ to the `main` procedure, i.e., we get

$pp \rightsquigarrow \text{ENTRY};\ \mathcal{B}[\,\Delta_b$ `in defs proc` $sort\ =\ \ldots$
`in` $\{\ [\,0\ 0\,]_b\ := <\ 5\ 4\ 9\ 8\ 1\ 6\ 3\ 2\ 7\ >;\ (\ [\,0\ sort\,]\ \&[\,0\ 0\,]_b\ \ 10\ );\ \ldots\ \mid 0\,];$
EXIT; .

```
main = Δb in
       defs
          proc sort =
             Δs *Δs in
                Δb in
                   defs
                      proc bubble =
                         Δs *Δs in
                            Δb in
                               defs
                                  proc swap =
                                     Δs Δs in
                                        Δb in { [ 0 0 ]b := *[ 1 0 ]s | [ 0 1 ]s;
                                                *[ 1 0 ]s | [ 0 1 ]s :=
                                                          *[ 1 0 ]s | [ 0 0 ]s;
                                                *[ 1 0 ]s | [ 0 0 ]s := [ 0 0 ]b
                                              }
                                  in{ if ( gt [ 0 0 ]s [ 1 1 ]s )
                                      then [ 0 0 ]b := ( + 1 [ 0 1 ]s );
                                           if ( lt *[ 0 0 ]s | [ 0 1 ]s
                                                             [ 0 0 ]s | [ 0 0 ]b )
                                           then ( [ 0 swap ] [ 0 1 ]s [ 0 0 ]b )
                                           else nop ;
                                           ( [ 1 bubble ] [ 0 0 ]b *[ 0 0 ]s )
                                      else ( [ 2 sort ] ( − 1 [ 1 0 ]b ) *[ 0 0 ]s )
                                    }
                            in { if ( gt 1 [ 0 1 ]s )
                                 then [ 0 0 ]b := 1;
                                      ( [ 0 bubble ] [ 0 0 ]b *[ 0 0 ]s )
                               else nop
                               }
          in { [ 0 0 ]b := < 5 4 9 8 1 6 3 2 7 >;
               ( [ 0 sort ] &[ 0 0 ]b 10 )
                 ...
             }
```

Fig. 13.7. The preprocessed version of the IL bubble-sort program of Fig 13.2

calling in turn rule (2). This rule compiles first the statement block of the `defs`, using again scheme $\mathcal{B}$ to generate the first piece of code of Fig. 13.8, to which *pp* points. It also creates a heap structure for the argument array referenced by the pointer *p_vec* (second line of Fig. 13.8).

The compiler then turns to the procedure *sort*, using rule (4), (3) and (9) of $\mathcal{B}$ and rule (13) of $\mathcal{C}$ in this order, to generate the top line of code for its body, to which *p_sort* points. This piece of code evaluates the predicate (`gt` 0 *n*) of the `if_then_else` clause and, depending on its value, branches either to the consequent or alternative code referenced by the pointers *p_t*0 and *p_f*0, respectively (third line of Fig. 13.8).

$pp \rightsquigarrow$ ENTRY; ALLOC 1; PUSH *p_vec*; ASS_L 0 1;
PUSH 10; COPY 0 1; LINK 0 2; BSR *p_sort*; FREE 2; EXIT;

$p\_vec \rightsquigarrow$ < 5 4 9 8 1 6 3 2 7 >

$p\_sort \rightsquigarrow$ ALLOC 1; COPY 0 1; PUSH 1; GT; BRC *p_t*0 *p_f*0; FREE 1; RET;

$p\_t0 \rightsquigarrow$ PUSH 1; ASS_L 0 4; COPY 0 1; COPY 0 4;
LINK 0 2; BSR *p_bubble*; FREE 2; RET;

$p\_f0 \rightsquigarrow$ NOP; RET;

$p\_bubble \rightsquigarrow$ ALLOC 1; COPY 1 2; COPY 0 2; GT; BRC *p_t*1 *p_f*1; FREE 1; RET;

$p\_t1 \rightsquigarrow$ COPY 0 2; PUSH 1; ADD; COPY 0 1; COPY 0 4; ADD; FETCH;
COPY 0 1; COPY 0 2; ADD; FETCH; LT; BRC *p_t*2 *p_f*2; RET;

$p\_f1 \rightsquigarrow$ ALLOC 1; COPY 0 1; COPY 1 4; PUSH 1; MINUS;
LINK 2 2; BSR *p_sort*; FREE 2; RET;

$p\_t2 \rightsquigarrow$ ALLOC 1; COPY 0 4; COPY 0 2;LINK 0 2; BSR *p_swap*; FREE 2; RET;

$p\_f2 \rightsquigarrow$ NOP; RET;

$p\_swap \rightsquigarrow$ ALLOC 1; COPY 1 1; COPY 0 2; ADD; FETCH; ASS_L 0 4;
COPY 1 1; COPY 0 1; ADD; FETCH;
COPY 1 1; COPY 0 2; ADD; STORE;
COPY 0 4; COPY 1 1; COPY 0 1; ADD; STORE; FREE 1; RET;

Fig. 13.8. The compiled IAM code of the bubble-sort program

Compilation of the rest of the program, i.e., of the procedures *bubble* and *swap*, follows the same routine recursively. The special rules (7a) and (11a) that apply to indexed accesses of array elements enter the game when the code for the body of *swap* is generated.

What looks like rather sophisticated code with several nesting levels for a fairly simple problem is primarily a consequence of the limited expressive power of IL, of the chosen programming style, to a lesser extent of the choice of the IAM instruction set, and of the schematic compilation rules which do not include any code optimizations. In the absence of loop constructs that belong to the standard repertoire of imperative languages, traversing through a sequence of indices, e.g., to inspect and update the elements of an array step by step, must be realized in IL by means of tail-recursive procedures that involve a lot of parameter passing through the stack (rather than incrementing index variables in place). Moreover, defining the nonrecursive procedure *swap* locally to the procedure *bubble* is a somewhat artificial choice to create more than one nesting level and, in consequence, more activity on the stack. This

situation could be easily avoided by inlining the code of *swap* where it is called in the code of *bubble*, which could conveniently be done either in the IL source program or figured out by a code optimizer. This would eliminate almost the entire piece of code referenced by p_f1, which creates in the stack the argument frame for *swap*, then calls the procedure and, upon returning, clears the frame again. With a more suitable conditional branch instruction such as JFALSE of the machines described in Chap. 10, the do-nothing codes for the alternatives referenced by the pointers p_f0 and p_f2 could also be eliminated.

13.7 Outline of a Machine for a 'Flat' Language

One of the important flavors of imperative languages of the IL variety, e.g., PASCAL or MODULA, is the freedom to define procedures local to others. It allows a very structured approach toward program design in which the declaration of procedures, or of variables for that matter, may be carefully confined to exactly the contexts in which they are needed. Local procedures may be open in the sense that their bodies may contain occurrences of relatively free variables declared in enclosing contexts, which in fact justifies the nesting of procedures in the first place. The scoping of variables follows the λ-calculus insofar as the scopes of `sub`- and `bind`-bound variables extend over the respective procedure bodies, the scopes of `defs`-bound identifiers extend over the entire `defs` expressions, and outer bindings are shadowed against inner bindings to the same variable names.

However, the elegance of program design that comes with this flavor, as we have seen, must be paid for at the machine level with an environment structure in which activation records held in a stack must be linked up by chains of static pointers that reflect the nesting of procedure definitions in the source program. Both linking newly created records correctly to existing environments and accessing environment entries generally requires dereferencing pointers along these chains, followed by other operations on the stack, which in a real machine involves fairly time-consuming address computations.

However, we may get rid of these difficulties by switching from open to closed procedures. It renders the nesting of procedures and, in consequence, the linking of their activation records superfluous. This can be had either by ruling out nested procedures in the source language or by transforming, before compilation to machine code, programs of the IL language variety into some intermediate language that supports only closed procedures.

This approach is in fact similar to the concept of supercombinators outlined in Chap. 9. Section 9.1 tells us how to transform open into closed abstractions (or supercombinators) by λ-lifting relatively free variables to the next higher level of abstractors, and Sect. 9.2 shows that the environments for executing supercombinator codes are contained in coherent stack frames that, other than for the order in which they are called, are completely unrelated

to other frames held in the stack, i.e., there are no static links among them. There is a small price to be paid for this simplification, though. It comes in the form of additional parameters that need to be passed to procedures from which relatively free variables have been lifted.

Well-known languages that support only closed procedures and, interestingly enough, are in widespread use are C, C^{++} and to some extent also JAVA.[9]. Specifically C has become increasingly popular both as a programming language and as a standard intermediate language shared by compilers of other high-level languages to take advantage of highly optimized target code generation techniques supported by machine-specific C-compiler backends. Some of these code optimizations benefit greatly from the fact that procedures are closed.

The syntax of a language kernel CIL that derives from IL but supports only closed procedures is defined in Fig. 13.9.

$$
\begin{aligned}
\mathit{program} &=_s \texttt{main} = \mathit{alternatives} \\
\mathit{alternatives} &=_s \texttt{bind}\ \mathit{vars}\ \texttt{in}\ \{\ \mathit{statements}\ \} \mid \\
&\quad\ \texttt{globals}\ \mathit{vars}\ \texttt{in}\ \texttt{defs}\ \mathit{procedures}\ \texttt{in}\ \{\ \mathit{statements}\ \} \mid \\
&\quad\ \texttt{defs}\ \mathit{procedures}\ \texttt{in}\ \{\ \mathit{statements}\ \} \\
\mathit{procedures} &=_s \mathit{procedure} \mid \mathit{procedures};\ \mathit{procedure} \\
\mathit{procedure} &=_s = \texttt{sub}\ \mathit{form\_pars}\ \texttt{in}\ \mathit{body} \\
\mathit{body} &=_s \{\ \mathit{statements}\ \} \mid \texttt{bind}\ \mathit{vars}\ \texttt{in}\ \{\ \mathit{statements}\ \}
\end{aligned}
$$

Fig. 13.9. The syntax of a flat imperative kernel language CIL that supports only closed procedures

In this syntax, a program is either a block of statements preceded by an abstractor (or binder) for variables used in it, or a `defs` construct for mutually recursive procedures which may or may not be preceded by an abstractor `globals` for global variables used in any of the procedures or in the statement block. The syntax of procedures is as defined in IL, except that their bodies are just blocks of statements that may or may not be preceded by abstractors for local variables. The important point is that these bodies, in contrast to IL, may not recursively contain further procedure definitions, which also implies that the variables that may occur in the statement blocks of procedures are either their `sub`-bound formal parameters, their `bind`-bound locals

[9] JAVA allows to define classes and, hence, procedures called methods that are local to each other. However, the JAVA virtual machine requires that inner classes be flattened by a preprocessor before compilation to code.

or `globals`-bound globals. Statements are defined exactly as in IL and are therefore not repeated in Fig. 13.9.

We refer to CIL as being a flat language since there is only one level of procedure definitions, i.e., a CIL program may be composed of at most one `defs` construct (possibly enclosed in a single `globals` definition), and the machinery that supports it is called a flat imperative abstract machine, abbreviated to FIAM.

The procedures, as may be noted, are closed only with respect to their `sub`- and `bind`-bound variables, but not with respect to global variables. However, this does not create much of a problem since the globals are shared among all the procedures declared in the program, including the top-level statement block, and may therefore be held in some fixed locations, preferably at the bottom of the stack, where they can be conveniently accessed from anywhere else. This goes hand in hand with the fact that space for these globals must obviously be allocated in the very first step of code execution.

A typical stack configuration of the FIAM that shows the situation after some procedure f has called a procedure g is depicted in Fig. 13.10. At the very bottom of the stack (or at the highest addresses) we find a frame for the instantiations of the program's global variables, with a pointer *p_global* pointing to the next to topmost location of this frame, relative to which the frame entries may be addressed. The activation records for f and g comprise argument frames and workspaces for the procedures parameters and local variables, and in between them the return addresses, just as in the IAM, but the link pointers have disappeared.

Since the code of an active procedure accesses `sub`- or `bind`-bound variable instantiations only in what is currently the topmost activation record, we may conveniently use the current stack-top pointer *ps* as the record's base, relative to which its entries may be addressed with positive offsets derived from binding indices. However, the compiler has to take into account the fact that this point of reference is not fixed but may change due to temporaries pushed onto and removed from the stack, which means that a particular record entry may have different offsets relative to the stack top, depending on the state of code execution (see also the description of the *G*-machine compiler in Sect. 9.3).

To figure out what the offsets in the activation records are, or where its entries need to be placed, we again engage a preprocessor to convert bound variable occurrences into index representations. However, since only closed procedures defined at the same level need to be dealt with, we can simply use plain binding indices again since we don't have to worry about nesting levels. We need only to distinguish between `globals`-bound variables on the one hand and `sub`- or `bind`-bound variables on the other hand since they must be sorted into two different bins – the former to access the globals frame at

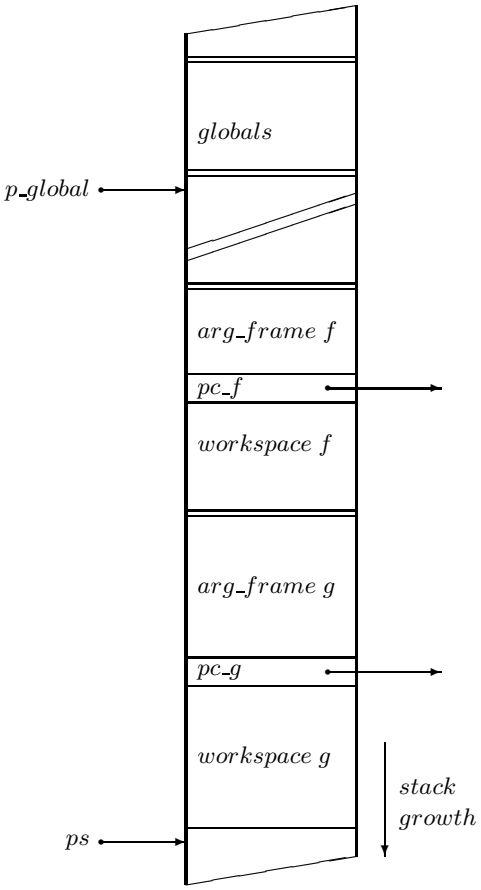


Fig. 13.10. A FIAM stack configuration with a procedure f having called another procedure g

the stack bottom, and the latter to access the activation record at the stack top.[10]

To illustrate how preprocessing basically works, consider the following fragmental CIL program:
where $h \in \{ 1, \ldots, k \}$, $i \in \{ 1, \ldots, m \}$ and $j \in \{ 1, \ldots, n \}$. It is converted into the index representation

[10] As in the case of the IAM, we also must of course distinguish between `sub`- and `bind`-bound variables of a procedure to get the offsets relative to the position of the return addresses within the activation record right.

$$
\begin{array}{l}
\texttt{globals}\ u_1\ \ldots\ u_k\ \texttt{in} \\
\quad \texttt{defs} \\
\qquad \ldots \\
\qquad \texttt{proc}\ pp\ =\ \texttt{sub}\ v_1\ \ldots\ v_m\ \texttt{in} \\
\qquad\qquad\qquad \texttt{bind}\ w_1\ \ldots\ w_n\ \texttt{in} \\
\qquad\qquad\qquad \{\ldots u_h \ldots v_i \ldots w_j \ldots\} \\
\qquad \ldots \\
\quad \texttt{in}\ \{\ldots\}\ , \\
\\
\underbrace{\Delta_g \ldots \Delta_g}_{k}\ \texttt{in} \\
\quad \texttt{defs} \\
\qquad \ldots \\
\qquad \texttt{proc}\ pp\ =\ \underbrace{\Delta_s \ldots \Delta_s}_{m}\ \texttt{in} \\
\qquad\qquad\qquad \underbrace{\Delta_b \ldots \Delta_b}_{n}\ \texttt{in} \\
\qquad\qquad\qquad \{\ldots \#_g(k-h-1) \ldots \#_s(m-i-1) \ldots \#_b(n-j-1) \ldots\} \\
\qquad \ldots \\
\quad \texttt{in}\ \{\ldots\}\ ,
\end{array}
$$

where the anonymous abstractor sequences $\underbrace{\Delta_g \ldots \Delta_g}_{k}$, $\underbrace{\Delta_s \ldots \Delta_s}_{m}$ and $\underbrace{\Delta_b \ldots \Delta_b}_{n}$ replace the binders $\texttt{globals}\ u_1\ \ldots\ u_k$, $\texttt{sub}\ v_1\ \ldots\ v_m$ and $\texttt{bind}\ w_1\ \ldots\ w_n$, respectively, and the binding indices $\#_g(k-h-1)$, $\#_s(m-i-1)$, $\#_b(n-j-1)$ replace the variable occurrences u_h, v_i, w_j, respectively, denoting their declaration positions.

Translating the binding indices $\#_s ii$ and $\#_b jj$ into offsets relative to the current stack top ps may be based on the following consideration: we know that the size of an activation record that is topmost on the stack and thus active is given by $size = m+n+s+1$, where m and n are the numbers of procedure parameters and local variables, respectively, s accounts for the number of temporaries on top of the workspace, and another entry accommodates the return address.

Following the convention that we have used in the IAM, the code for a procedure call ($proc\_name\ act\_par\_1 \ldots act\_par\_m$) is supposed to push the actual parameters in the order from right to left. That is to say, the argument to be substitued for the index $\#_s(m-1)$ is topmost, or at offset 1, and the argument to be substitued for index $\#_s 0$ is lowermost, or at offset m, relative to the top of a stack in which just the complete argument frame of an activation record has built up.[11] Relative to the full activation record these offsets must be shifted to $n+s+2$ and $m+n+s+1$, respectively. Thus, we may transform binding indices $\#_s ii$ into offsets relative to the stack pointer

[11] Remember that the stack top pointer ps points to the first empty position on top of the stack.

ps as follows:

$$off\_s(\ \#_s ii\ )\ =\ m+n+s+1-ii \mid ii \in \{\ 0,\ldots,\ m-1\ \}\ .$$

The same convention may be used with regard to the orientation of indices $\#_b jj$ in the locals frame of the workspace (whose upper and lower boundaries are at offsets $s+1$ and $n+s$) so that they transform as

$$off\_b(\ \#_b ii\ )\ =\ n+s-jj \mid jj \in \{\ 0,\ldots,\ n-1\ \}\ .$$

Things become even simpler for the globals indices $\#_g hh$, which transform as

$$off\_g(\ \#_g hh\ )\ =\ k-hh \mid hh \in \{\ 0,\ldots,\ k-1\ \}$$

into offsets relative to the globals frame pointer pg (see Fig. 13.10).

The instruction set of the FIAM is essentially the same as that of the IAM, except that the instruction LINK h k has become superfluous and that both the COPY and the ASSIGN instruction split up into two which come with only one index parameter j:

COPY_G j copies to the top of the stack the entry found at an offset j relative to the top pointer *p_global* of the globals frame;

COPY_S j copies to the top of the stack the entry found at an offset j relative to the stack top pointer ps;

ASSIGN_G j assigns the value taken from the top of the stack to the entry found at an offset j relative to the top pointer *p_global* of the globals frame;

ASSIGN_S j assigns the value taken from the top of the stack to the entry found at an offset j relative to the stack top pointer ps.

Compiling CIL programs to FIAM code follows a scheme that is similar to the compilation to IAM code. It is a little more complicated in that addressing stack entries relative to changing stack top pointers rather than fixed base addresses of activation records necessitates carrying along the four parameters $m,\ n,\ s$ and k by means of which binding indices must be converted into offsets. We thus have

$$\mathcal{Z}[\ s\_form : rest \mid (m,\ n,\ s,\ k)\ ] \Longrightarrow code[\ s\_form\ ];\ \mathcal{Z}[\ rest \mid (m,\ n,\ s,\ k)\ ]$$

as the general scheme for compiling syntactical forms s_form of CIL. Whereas the parameter k remains fixed throughout the entire compilation process, m and n change between procedure codes, and s changes whenever temporaries are pushed onto or popped off the stack.

13.8 Summary

In this chapter we have crossed the borderline between computing the values of expressions and performing computations for their (side)effects on a state. This imperative (or procedural) model of computation differs from the λ-calculus in that it

- downgrades the notion of λ-bound variables that are their own values (and thus are first-class objects) to that of box variables which represent changeable values (environment entries) but are not values themselves;
- introduces the notion of assignment statements that modify the values represented by the box variables and, closely related to this idea,
- introduces procedures as another kind of abstractions that effect changes in the values of box variables in their calling environments;
- outlaws, as a matter of convenience, not of necessity, partial procedure applications to simplify the implementation.

Procedure definitions may be arbitrarily nested, permitting variables to be free in local contexts but bound higher up (i.e., the procedures may be open), with the scoping rules remaining the same as in the λ-calculus.

The ensuing imperative abstract machine IAM closely resembles the SECD_I machine of the preceding chapter, except that it has merged the value, environment and dump stacks into a single runtime stack. This has become possible because the machine does not need to deal with closures representing partial applications, as a consequence of which activation records of procedure calls may be released strictly in reverse order of their creation. However, what is inherited from the SECD_I machine is the linking of activation records in accordance with the nesting of procedure definitions in the source program, and along with it the dereferencing of these link pointer chains to access entries deeper down in the environment structure. An activation record itself includes, in this order, a link pointer to the preceding record in the environment, an argument frame for instantiations of procedure parameters, a return address and a frame for local variables, followed by dynamically changing numbers of temporaries that together make up the workspace.

The IAM instruction set is nearly the same as that of the SECD_I machine. What is added are merely assignments that update environment entries.

The chapter has also shown that the machinery, and specifically the representation of environments in the stack, can be considerably simplified when switching to a flat source language that permits only closed procedures defined on the same level. Implementing this approach, which resembles the supercombinator concept of Chap. 9, disposes of the link pointer chains as the environments for procedure calls are, apart from a set of global variable instantiations, completely contained in single activation records. These records are stacked up in the order in which the procedures are called but otherwise are completely unrelated to each other.

References

The contents of this chapter have been adopted from the author's unpublished classroom notes.

Simple abstract machines for imperative languages usually serve as interfaces between compiler front- and backends. They come either in the form of tree languages whose instructions correspond to the nodes of syntax trees (see for instance [App99]) or as abstract stack machines with small instruction sets for arithmetic, stack handling and control flow (see for instance [ASU85]). These concepts are also described in other textbooks on compiler construction, e.g., in [Ma96, Wi96, Kap94].

Examples of language-specific imperative abstract stack machines are the Algol Object Code for ALGOL 60 [RR64], the PASCAL P4 machine [Amm81] for which a compiler is described in [PeDa82], and the UCSD P machine, also for PASCAL [ClKo82].

Prominent examples of object-oriented abstract stack machines are the JAVA virtual machine [LiYe99, SSB01] and the virtual SMALLTALK 80 machine [GoRo89].

Also in the category of imperative abstract machines fall those for string processing, say for the interpretation of shell commands or of script languages, to which references may be found in [DHS00].

14

Real Computing Machines

There is little difference conceptually between the various code-executing abstract machines described in the preceding chapters on the one hand and real computing machines on the other hand. What matters from a users point of view are their instruction sets, addressing modes and the resources visible at the machine language level, which are said to define the architectures of the machines. The visible resources basically include a processing unit equipped with an arithmetic/logic unit and a set of fast working registers (of which some are reserved for special purposes such as program counter and stack pointer), a uniformly addressable main memory, and various peripheral devices operated through dedicated input/output ports.

Programming at this level is, for good reasons, usually confined to a few system kernel routines, e.g., for interrupt handling, process scheduling and input/output operations, and to the implementation of compiler backends that generate executable code.

It is primarily the finiteness of resources and certain bandwidth limitations that must be taken into consideration in machine-level programming beyond what is specified by abstract machine code. Major concerns, on the one hand, are the efficient utilization of finitely many processor-resident registers for speedy program execution and the management in main memory of dynamically expanding and contracting structures such as stacks and heap space. On the other hand, fixed sizes of physical registers and addressable memory entries, measured in units of bits, bytes or words, render it necessary to squeeze instructions, data and addresses into fixed formats that can be readily decoded (or interpreted) by the underlying hardware machinery. Data formats determine the choice of instructions that may be used to operate on them, instruction formats determine the total number of instructions that can be supported, the number of operands and the operand addressing modes that may be used, and address formats determine the size of the effectively addressable main memory.

The architectures of real computing machines are dictated primarily by the needs of imperative programming languages, but beyond that they are also

driven to a considerable degree by technological progress. Steadily increasing miniaturization of electronic circuitry has led to rapidly growing capacities of cache, main and peripheral memories, and, even more importantly, to processor designs featuring sophisticated controls, large register sets and advanced pipeline processing. Computational speed has been increased by several orders of magnitude due to a combination of faster memory access cycles and higher clock frequencies at which the processors are driven.

Earlier processor generations that dominated the scene in the 1980s are characterized as complex instruction set computer architectures (CISCs for short) or as register/memory (R/M) architectures, of which the VAX/750, the MC68000 and the Intel 8080/8086 families are typical examples. Their instruction sets and addressing modes facilitate the compilation of high-level languages into fairly dense code for machines with limited memory capacities. The data-processing instructions come in different formats for the operand data and may be combined with a variety of addressing modes. Instruction decoding requires complex microcoded controls; executing the same basic instruction may take varying numbers of machine cycles, depending on the chosen addressing modes and address positions in memory. Operands may be addressed either in main memory or in processor-internal registers. As we will see in the following sections, CISC architectures are a very close match for the implementation of the various abstract machines we have studied in the preceding chapters.

These architectures were superseded in the 1990s by reduced instruction set computer (or RISC) architectures. This development paid tribute to the fact that only about 30 of the more than 200 instructions of a CISC were used by compilers to generate machine code, and also of technological advances that led to the integration of entire processors on single chips. The first of these processors favored simple instruction sets suitable for pipelined execution at the rate of about one instruction per machine cycle.[1] All instructions have the same (word) format to facilitate decoding, all data-processing instructions use processor-internal registers only as operand sources and result destinations, and there are special instructions that load data from memory into registers and store data from registers into memory; RISCs are therefore also said to have a load/store architecture. Compilation and specifically code generation are more difficult than in CISCs since a larger gap has to be bridged between high-level languages and machine code, and some amount of code reorganizing is generally necessary to obtain the best possible pipeline performance. Code density is decidedly lower since more of the simpler RISC instructions are necessary to do the same jobs as CISC instructions. Typical examples of RISC processor architectures are the SPARC and MIPS families.

[1] It is of course assumed here that pipelining is supported by fast cache memories for both instructions and data and that a continuous flow of instructions through the pipeline can be sustained.

More recent trends in processor architectures, driven primarily by further technological advances and by the demand for still higher performance, are aimed at drastically increased instruction throughput. Superscalar processors run several instruction pipelines fed from a single instruction stream. The instructions become more complex again but are by predecoders converted into RISC-like operations for pipelined execution, as for instance in the Intel Pentium 4. Very Long Instruction Word (VLIW) processors execute in parallel the equivalent of several RISC-like instructions issued in one step. The responsibility for filling the slots in such instruction words rests entirely with the compiler. An example of such a processor is the Intel Itanium IA 64 that works with 128 bit instruction bundles, of which two combined may control execution of up to six RISC instructions.

However, as these advanced architectures are CISC or RISC at the core, we will focus in the following on representative classical machines of either kind to expose similarities and essential differences with regard to machine level programming. To this end, we will use the symbolic notation commonly used in assembler programming but largely ignore the details of instruction and data formats beyond their lengths (numbers of bits or bytes), and also ignore the intricacies of instruction execution that specifically in CISC processors are quite complicated. Little needs to be known about the underlying hardware machinery other than that there is a byte- or word-addressable main memory, a finite set of processor-internal working registers, dedicated registers for the program counter and a single stack pointer, and some arithmetic/logic unit that somehow does all the value-transforming and address computations.

Of RISC processors it suffices to know that the instructions must be simple enough so that their execution can be conveniently partitioned into four or five different phases that take about the same time to complete. These phases typically include instruction fetching (from an instruction buffer or a cache memory), decoding and data fetching (usually from a separate data cache), performing the specified operation, and writing the result back to cache. They correspond to successive pipeline stages that an instruction has to pass through step by step. The pipeline may process simultaneously as many instructions as there are stages, ideally producing a throughput of one instruction per unit time. In reality, however, throughput is lower than that since on average one in four instructions executed is a branch instruction that disrupts the continuous flow of instructions through the pipeline.

14.1 A Typical CISC Architecture

The architecture described in this section closely resembles that of the MC680x0 processor family, but we have taken the liberty of skipping or simplifying a few things that are not too relevant with regard to the basics of machine-level programming and program execution.

The characteristic features of this architecture are a relatively small set of just 16 processor-internal registers, a rich variety of addressing modes that facilitate creating, deleting and accessing runtime stack entries and heap objects held in the memory, and the freedom to combine these addressing modes with almost all data-transporting, data-processing and control instructions. The very basic instructions necessary to move data in memory, between memory and registers and about registers, to do primitive operations, and to excercise control over the sequencing of instructions are essentially the same as those used in the abstract machines of the preceding chapters. Mapping abstract machine code, at least that of the IAM or FIAM, to CISC-code is, apart from a few details pertaining to addressing and formats, more or less just a matter of transliteration.

14.1.1 The Register Set, Formats and Addressing in Memory

The register set, also called the register file, of this machine partitions into eight data registers named $d0 \ldots d7$ and eight address registers named $a0 \ldots a7$. By convention, the data register $d0$ is reserved for return values of function calls, address register $a7$ is reserved for the runtime stack pointer sp, and address register $a6$ accommodates the static link pointer to the environment of the active procedure call.The data registers $d1$ to $d7$ are mainly used for temporaries, i.e., the items that are held on the value stack S in our various abstract machines and, space permitting, also for frequently accessed locals. The address registers $a0$ to $a5$ may be used to hold various pointers to data structures held in memory, e.g., to additional stacks or heap locations that come into play when emulating one of the abstract machines described in the preceding chapters. Unused address registers may alternatively store data, but data registers may not be used for addressing.

An additional address register is provided for the program counter pc, and there is also a four-bit condition code register ccr signaling the outcome of all but the control instructions. These bits are denoted as z for zero results, n for negative results, c for carry conditions, and v for result overflows of arithmetic and compare operations. They replace the Boolean values deposited on the value stacks of our abstract machines for inspection by subsequent conditional branch instructions.[2]

Numerical data values may come in byte, halfword (2 bytes), word (4 bytes) or double-word (8 bytes) formats, character strings may have a length

[2] It should be noted that these are the registers that are accessible when the machine is in a so-called user mode, under which it executes code obtained by compilation of high level user programs, using a restricted set of nonprivileged instructions only. A few more registers and a few more privileged instructions become available under a supervisor or kernel mode to which the machine switches whenever system calls or interrupt conditions need to be dealt with. However, this mode is not of interest here.

of up to 256 bytes. In memory, the numerical formats are stored at byte addresses that are multiples of their sizes, in data registers they are aligned 'to the right', meaning that byte and halfword formats occupy the least significant byte positions. Double-word formats take two consecutive registers (with the least significant word in the evenly enumerated register). Character strings may begin at byte addresses. Depending on the number of operands and the addressing modes used, instruction formats may have one or several halfwords lengths; consecutive instructions are held under consecutive addresses in memory, without any gaps in between. The program counter traverses instruction formats in increments of halfwords under the control of the instructions themselves. As in the (F)IAM, the runtime stack grows toward lower addresses in decrements corresponding to the formats of the data entries pushed (and shrinks toward higher addresses in increments corresponding to the formats of the data popped).

A typical memory layout in the logical (or virtual) address space $[\,0 \ldots size-1\,]$ of an imperative program is depicted in Fig. 14.1.[3] It has the program code and static data located at the higher end of the address space. The dynamic parts, which include the runtime stack and the heap, occupy the lower end. As indicated by the horizontal arrows, the stack grows toward lower addresses, and the heap grows basically toward higher addresses, but may have holes that are left over by deallocated heap objects filled again with newly created objects of fitting sizes (the holes and the unused space between the stack and the heap are depicted as shaded areas).

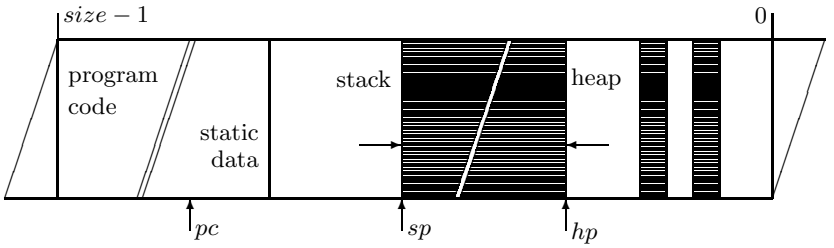


Fig. 14.1. Typical memory layout for executing an imperative program

There are basically three different ways of mapping logical address space into the real address space of a computer's memory. The brute force method allocates real address space in units of regions large enough to have entire logical address spaces loaded, with the consequences that (1) entire regions must be swapped between backup and main memory whenever a program run must be temporarily suspended and that (2) generally other regions must be

[3] The size of the logical address space is usually chosen to be some convenient value $n * 2^k$ large enough to accommodate the static and dynamic parts of a program run.

allocated for reloading. A more appropriate method has the logical address space partitioned into segments of variable size, say for pieces of code, the runtime stack and the heap, and has these segments placed into different parts of the real address space. Alternatively, the logical and real address spaces may be partitioned into equally sized page frames so that any logical page can be mapped into any real page, and only what is called a working set of pages needs to be kept in main memory at any time. Segmentation and paging may even be combined. Real address space can thus be managed more economically, and swapping segments or pages in and out of main memory may take less time (and possibly be hidden behind other activities) than swapping entire regions.

However, swapping inevitably requires that things be relocatable in real address space. This, in turn, means that they must be addressable with fixed offsets relative to suitable changeable base addresses, e.g., stack entries relative to the stack pointer *sp*, heap objects relative either to the heap base 0 or to some top-of-the-heap pointer *hp*, and program code relative to the program counter *pc*, which specifically applies to branch addresses. Once the respective address registers are loaded with the initial real addresses, or logical addresses are dynamically mapped to real base addresses by means of tables, code execution may proceed without updating any logical address specifications.

14.1.2 Addressing Modes

A distinguishing feature of our MC680x0-like CISC machine is the various addressing modes available and the freedom to combine them with almost all instructions. As stated before, these addressing modes are tailored to the needs of creating and accessing entries in memory-resident data structures such as stacks, linked lists of (compound) data, arrays, etc. They include addressing by

- direct specification of a register as the source or destination of a data item or of a memory address;
- implicit references, as part of the encoding of an instruction, to special registers such as the stack pointer or the program counter;
- computation of operand or branch addresses from base addresses, offsets, increments, decrements, indices and levels of indirection.

The addressing modes that are of interest here are summarized in Fig. 14.2. It lists, from left to right, the name of a mode, the syntax used in machine-level (or assembler) programming, and its effect (or interpretation). Using the notations *rand* for an operand value, *addr* for a memory address, *an* and *dn* for the contents of the registers involved, $MM[\ an\ ]$ for the contents of a memory location addressed with the contents of *an*, and $\rightleftharpoons$ symbolizing the reading (or writing) of the value on its left from (or to) the contents of what is specified on the right, we have from top to bottom the modes

mode	syntax	effect
`reg_direct`	dn an	$rand \rightleftharpoons dn$ $address \rightleftharpoons an$
`reg_indirect`	(an)	$rand \rightleftharpoons MM[\, an \,]$
`reg_indir_pre_incr`	$+(an)$	$an \leftarrow an + size$ $rand \rightleftharpoons MM[\, an \,]$
`reg_indir_post_decr`	$(an)-$	$rand \rightleftharpoons MM[\, an \,]$ $an \leftarrow an - size$
`reg_indir_offset`	$dis(an)$	$rand \rightleftharpoons MM[\, an + dis \,]$
`reg_indir_index`	$dis(an, xm)$	$rand \rightleftharpoons MM[\, an + xm + dis \,]$
`immediate`	$\#number$	$rand \leftharpoonup \#number$

Fig. 14.2. The addressing modes

`reg(ister)_direct` which reads or writes an operand value in a register dn or an address in a register an;

`reg(ister)_indirect` which reads or writes the value in memory location an;

`reg(ister)_indir(ect_with)_pre_incr(ement)` which first increments the contents of the address register an by the size of an operand specified as part of the instruction that uses the mode, and then does the same as the mode `reg_indirect`;

`reg(ister)_indir(ect_with)_post_decr(ement)` which does the same as the mode `reg_indirect` and then decrements the address in an, again by the size of an operand specified as part of the instruction that uses the mode;

`reg(ister)_indir(ect_with)_offset` which reads or writes the memory location obtained by adding to a base address in an an offset (or displacement) dis that may be positive or negative;

`reg(ister)_indir(ect_with)_index(ing)` which reads or writes a memory location obtained by adding to a base address in an an index contained in another address register xm and an offset dis;

`immediate` which introduces as a direct operand a constant numerical value $\#number$ that may, for instance, be written into a value register or into a memory location.

The `reg_indir_offset` and `reg_indir_index` modes are also applicable to the program counter pc instead of one of the address registers an.

Of particular interest with regard to our various stack-based machines are the modes `reg_indir_pre_incr` and `reg_indir_post_decr` which may be used to respectively increment the stack pointer prior to reading an item from the stack, thus realizing a pop operation, and to decrement the stack pointer after having written something onto the stack, thus realizing a push operation. As these modes can be used with any of the address registers, not just with the default stack pointer $a7$, there is ample opportunity to support several stacks.[4] Equally important is the mode `reg_indir_offset` which may be used to access stack entries relative to the stack top or to any other fixed stack address, e.g., relative to some suitably chosen base of an activation record, and to branch, relative to the current program counter, to an instruction address other than the next one in sequence.

Dereferencing an address k times, as is necessary to access environment entries in activation records or argument frames k nesting levels down from the top, may be accomplished by having the `reg_indir` mode applied by a sequence of k instructions that move the contents of some memory location $MM[\, an \,]$ into the register an.

14.1.3 Some Important Instructions

There are only a few instructions that we need to know about in order to be able to implement our various abstract machines on this MC680x0-like CISC architecture.

The control instructions include

BSR dis which branches unconditionally to procedure code located a distance of dis bytes (positive or negative) away from the current progam counter position pc. It does so by first saving as return address the current pc on the stack, then decrementing the stack pointer by four byte positions (the length of the return address), and finally setting the program counter to $pc + dis$, i.e., it does the steps $(sp) \leftarrow pc$; $sp \leftarrow sp - 4$; $pc \leftarrow pc + dis$.

B_cc dis which branches conditionally to code located a distance of dis bytes away from the current pc, the condition being specified by the subscript _cc which may assume one of the values z, n, c, v or nz, nn, nc, nv (for negation of the former) of the condition codes set by preceding instructions. If the condition is not satisfied, code execution continues with the next instruction in sequence. Thus, the effect of the instruction may be specified as `if` $cc = true$ `then` $pc \leftarrow pc + dis$ `else` pc.

RTD dis which returns from a procedure call by taking the return address off the stack and subsequently incrementing the stack pointer by the offset dis to deallocate parameters from the stack, i.e., the sequence of steps is $sp \leftarrow sp + 4$; $pc \leftarrow (sp)$; $sp \leftarrow sp + dis$.

[4] These two modes may also be (mis)used to traverse array structures, instead of using the `reg_indir_index` mode.

RTS which is the simpler version of RTD that does not pop parameters off the stack.

Both branch instructions have their addressing modes implicitly defined as `reg_indir_offset`, using the register *pc* instead of one of the address registers *an* as the base of the offset. Likewise, both return instructions implicitly refer to the stack pointer as the address register that needs to be operated on.

Another two control instructions handle the static link pointers that connect activation records of procedure that are local to each other.

LINK *an dis* establishes a link between a newly created activation record of a procedure call and its environment. It does so by first saving on the stack the current environment pointer held in *an*, then overwriting *an* with the current stack pointer, and finally adding the displacement *dis* to the stack pointer, i.e., it performs the steps $(sp) \leftarrow an;\ an \leftarrow sp;\ sp \leftarrow sp - 4 + dis;$

UNLINK *an* is complementary to LINK. It deletes the topmost activation record from the environment, and it does so by updating the stack pointer with the contents of the register *an* and then overwriting the register *an* with the pointer to the rest environment that it pops off the stack, i.e., we have $sp \leftarrow an;\ sp \leftarrow sp + 4;\ an \leftarrow (sp);$.

Another important class of instructions are those that copy the contents of registers or storage locations. These are instructions that must have specified the format of the data items to be copied. This is done by attaching to the instruction's name a label .F which may assume any of the values .B (for byte), .H (for halfword), .W (for word), .D (for double word) and .S (for character string). The copy instructions of interest are the following:

MOVE.F *source dest* copies a data format .F from a source location *source* to a sink location *dest*, both are specified by any of the addressing modes listed in Fig. 14.2;

MOVE.A *source dest* copies an address from a source location *source* to a sink location *dest*, again both are specified by any of the addressing modes of Fig. 14.2, with the restriction that source or destination must be an address register when using the `reg_direct` mode;

MOVEM *reg_list* $(sp)-$ pushes multiple data and address registers from the list $reglist =< r_{i1} \ldots r_{ik} >$ [5] into the stack in the order from left to right; conversely, MOVEM $+(sp)$ *reg_list* loads from right to left the registers listed in *reg_list* with data or addresses taken off the stack;

LEA *source an* loads the register *an* with the address specified by *source*.

The instructions that do the actual computing look very much the same syntactically as the MOVE instructions. They are two-address instructions, with the second address specifying both the source of the second operand

[5] In symbolic assembler code, the registers may be specified by their names, at the machine level this list translates into a bit vector of 16 positions, with $r_j = 1$ if the j-th register is to be moved, and with $r_j = 0$ otherwise.

and the destination of the result, the preferred addressing mode for it being `reg_direct`. Typical examples are the instructions

ADD.F *source_1 dn* which adds two integer data values of format .F and writes the result to data register *dn*;
ADD.A *source_1 an* which adds two addresses (of word format) and writes the resulting address to address register *an*;
CMP.F *source_1 dn* which compares two data values of format .F;
CMP.A *source_1 an* which compares two addresses.

All arithmetic and relational instructions set condition codes which may become relevant to subsequent instructions that test them. This is specifically the case with the CMP.F instructions where the condition codes are the results that really matter since they are usually inspected by conditional branch instructions that follow next in sequence.

14.1.4 Implementing Procedure Calls

Given the definition of the instructions LINK and UNLINK, procedure calls may in our *CISC* machine be implemented in about the same way as in the IAM described in Sect. 13.3.1. The layout of the activation records, specifically the placement of the static link pointers, closely resembles that in Fig. 13.3, with a slight difference though. Since the processor includes a set of registers for temporary data and addresses, some of which may be used by a procedure in some state of execution, provisionxs must be made to save register contents in the stack when another procedure is called. These may be the registers used either by the calling or by the called procedure, which must be restored (or unsaved) upon returning. The most appropriate place in an activation record for saving these registers is the one immediately following the return address to make sure that they are cleared before the callee does anything else.

Another minor difference relates to the positions in the activation records of the static and dynamic links which, due to the particularities of the instructions LINK and UNLINK, must be interchanged.

A complete activation record for some procedure f is shown on the left of Fig. 14.3. Between the argument frame of f and the registers that need to be saved it features a control block whose entries are, in this order, the dynamic link old_a6 to the activation record of the calling procedure, the static link pe_f to the environment of f, the current program counter value pc_f, and some status register $status\_f$ that includes the condition code cc. The contents of the register $a6$, which points to the stack entry that contains the static link pe_f to the environment of f, may be used as a base address, sometimes also referred to as a frame pointer, relative to which all other record entries may be conveniently accessed with fixed offsets.

The first steps of procedure f calling another procedure g consist of first pushing the arguments of g and then executing the instruction

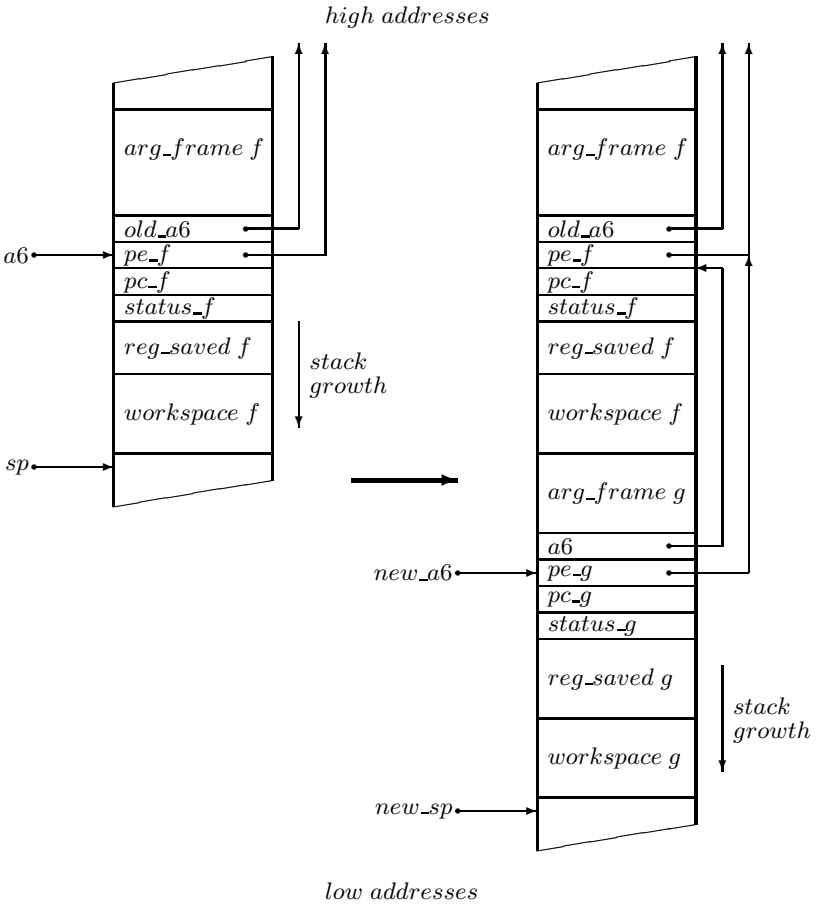


Fig. 14.3. Stack configurations before and after a procedure f calls a procedure g defined at the same nesting level

$$\text{MOVE.A } a6\ (sp)- \ ,$$

which saves the actual contents of $a6$, i.e., the pointer to the static link pe_f, by pushing it onto the stack as well, thus establishing the dynamic link to the activation record of f.

As we know from Sect. 13.3, the following cases need to be distinguished with regard to the static linking of the called procedure g: if it is defined

- to be local to procedure f, then $a6$ points to its environment and nothing needs be done;
- on the same or some $k > 0$ levels above procedure f, then $a6$ must be dereferenced by executing $k + 1$ times in succession the instruction

$$\text{MOVE.A } (a6)\ a6$$

to update $a6$ with the environment pointer pe_g of the callee g.

The second step completes the linkage by executing the instruction[6]

$$\text{LINK } a6\ \#0\ ,$$

which saves on the stack the new environment pointer pe_g held in register $a6$ and then updates $a6$ with the actual stack pointer sp.

The stack configuration on the right of Fig. 14.3 shows how the activation record of g thus links up if g is defined at the same nesting level as f, i.e., both share the same environment. In this particular case, the instruction MOVE $(a6)$ $a6$ must be executed just once so that pe_g (to which the updated contents new_a6 of $a6$ are pointing) and pe_f become the same. The stack entry $a6$ underneath pe_g is the dynamic link to the static link entry pe_f of the caller's activation record.

The entire linkage, i.e., the instruction sequence

$$\text{MOVE.A } a6\ (sp)-;\ \text{MOVE.A } (a6)\ a6;\ \text{LINK } a6\ \#0\ ,$$

is performed by the calling procedure f immediately before it executes the BSR instruction that branches to the procedure g, and so is the unlinking by means of the instruction UNLINK that must immediately follow the branch instruction.

Figure 14.4 shows how the piece of code of a procedure f that surrounds the call of another procedure g basically looks like. It begins with n MOVE.F instructions that push n arguments from the source addresses $source\_1\ \ldots\ source\_n$ onto the stack, followed by the instruction sequence that creates in the stack the static and dynamic links, which may require several successive MOVE.A instructions to deference the current static link. Immediately after the BSR instruction, the linking is undone and the argument frame is released simply by resetting the stack pointer by the frame size.

The code of the procedure g begins at *label* with a prelude that first saves the status register and the working registers to be used and then allocates workspace by advancing the stack pointer by the required size. Following the code that actually does the computing, the prelude is reversed to deallocate the workspace and restore the saved registers, before control returns, by means of the instruction RTS, to the caller f.

Essentially the same linking mechanism may be used when the static (and dynamic) links are placed underneath the activation records rather than between their argument frames and the return addresses of the procedure calls. However, since this means that the caller has to do the linking before setting up the callee's argument frame, we have to be careful about the register $a6$. If it is used as the base address relative to which all frame entries are accessed with fixed offsets, it cannot be updated before the argument

[6] $\#val$ denotes an immediate value which in this particular case adds the value 0 to the stack pointer sp.

```
        ENTRY;
        ...
        MOVE.F source_1 (sp)−;
        ...                                  /* pushing n arguments
        MOVE.F source_n (sp)−;
        MOVE.A a6 (sp)−;
        MOVE.A (a6) a6;
        ...                                  /*doing the links
        MOVE.A (a6) a6;
        LINK a6 #0;
        BSR label;                           /*calling procedure g
        UNLINK a6;                           /*undoing the links
        ADDA #size_of_arg_frame sp;          /*deleting the arguments
        ...
        EXIT;


label : MOVE.W status_reg (sp)−;             /*saves status register
        MOVEM reg_list (sp)−;                /*saves registers
        SUB.A #size_of_workspace sp;         /*allocates workspace
        ...
        ...                                  /*code of g statement block
        ...
        ADD.A #size_of_workspace sp;         /*deallocates workspace
        MOVEM +(sp) reg_list                 /* restores saved registers
        MOVEM +(sp) status_reg               /*restores old status
        RTS                                  /*returns to calling procedure
```

Fig. 14.4. The basic code structure of a procedure f that calls a procedure g defined at the same nesting level

frame has been completely built up, as otherwise at least the source addresses $source\_1 \ldots source\_n$ of the arguments, and possibly other address computations in between, may get corrupted. To play it safe, we therefore need to do the dereferencing of the static links, if necessary, on another address register, say $a5$, which changes the sequence of instructions that handle the links to

```
MOVE.A a6 (sp)−; MOVE.A a6 a5;
                 MOVE.A (a5) a5; ...; MOVE.A (a5) a5; LINK a5 #0;   .
```

Next, the arguments may be pushed onto the stack using the old contents of $a6$ as the base of their source addresses, and then the new static link held in $a5$ may be copied to $a6$, using the instruction MOVE.A $a5$ $a6$ immediately before the branch instruction BSR.

Figure 14.5 again shows the stack configurations before and after a procedure f calls a procedure g, with both f and g being defined at the same nesting level. On the left we see the environment link pe_f for f underneath

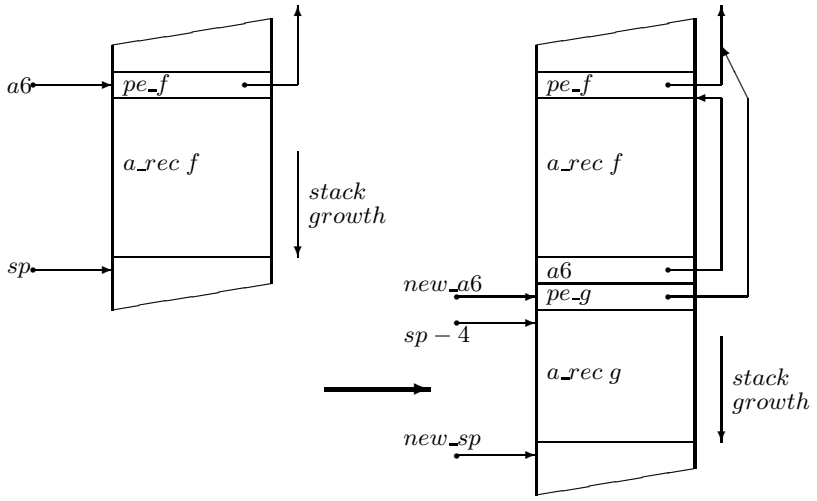


Fig. 14.5. Linking the activation record of a procedure call, with the link pointer entries placed underneath the record

the activation record *a_rec f*, with the contents of *a*6 pointing to this link entry and the stack pointer *sp* pointing to the top of *a_rec f*. After the above instruction sequence has been executed, the arguments for procedure *g* have been stacked up, and the link pointer in *a*5 has been copied back to *a*6, the register *a*6 is updated to *new_a*6, and the old contents of *a*6 are underneath. At this point, the activation record of *g*, though not yet complete, is fully linked up to its environment. Completing the record *a_rec g* moves the stack pointer further down to the position *new_sp*.

The calling procedure *f* again does all the linking, and also the unlinking, once control has returned from the procedure *g* and the stack has been cleared again by the procedure *f* down to the position *sp* − 4. Executing in this configuration the instruction

UNLINK *a*6

reconstructs the configuration on the left of Fig. 14.5.

All entries within an activation record may now be accessed with negative offsets relative to the link pointer, i.e., relative to the actual contents of the register *a*6, just as in the IAM described in Sect. 13.3.2.

14.2 A Typical RISC Architecture

One of the most challenging problems of compiling to real machine code is the efficient use of the processor-internal working registers. As stated before, they

take over the role of the value stacks of our various abstract machines, i.e., they accommodate primarily temporaries but, if available in excess to that, may also be allocated to local variables or even to procedure parameters.

The trouble with these register sets is that, in contrast to a stack of (conceptually) unlimited depth, the number of data items that can be held there is fairly small, particularly in the CISC architecture of the preceding section. Moreover, the register indices may be used only as address components of instructions but they cannot be placed into other registers or memory locations, as is the case with memory addresses, i.e., there can be no indirect addressing and hence no linking across registers. These constraints create a considerable register allocation problem if the objective is to keep as many active data as possible resident in registers, and to minimize data traffic between the registers and main memory.

Register allocation is usually based on the idea of coloring of what arei called interference graphs that derive from a program's flow of control and data. The nodes of these graphs represent temporaries (or locals), and edges between the nodes identify those items that cannot be placed into the same registers because they are used in the same phases of program execution. The nodes are then colored so that no two nodes connected by an edge receive the same color. If there are just as many colors needed as there are registers, then they constitute a valid register assignment that is free of interferences. Otherwise, as many data items as there are colors exceeding the number of registers need to be moved out and placed into memory.

Actually, register allocation is a little more involved than that but the coloring method is the basic idea. The problem becomes less severe with growing numbers of processor-internal registers. When more data is held in registers, there is less data traffic from and to memory. As computations are usually performed within a slowly shifting, fairly narrow scope of data items, most of the operations can then be done register to register, which helps to speed things up considerably. In fact, load/store instructions which cleanly separate data traffic to and from the memory from computations, large register sets, instruction pipelines and separate instruction and data caches are the major performance-enhancing components of RISC processor architectures.

14.2.1 The SPARC Register Set

There are basically two alternative ways to implement a register file.

The simplest one is a flat array of some 2^k registers, enumerated $0 \ldots 2^k - 1$, as for instance in the Intel 88110 and in the IBM Power PC processor architectures. However, with k approaching 8 it may become increasingly difficult to fully utilize all registers in a given computational context, say of a procedure call, unless code optimization techniques such as inlining and unrolling are rigorously applied. Beyond that there are two more reasons that argue against excessively large register arrays. One the one hand, the binary representation of register addresses increases logarithmically with the array size, which may

cause problems with the formatting of word-sized (32 bit) instructions. On the other hand, context switching, which requires that all register contents be swapped, becomes increasingly more costly.

A more structured approach with regard to the organization of a register file can be found in the SPARC architecture, at which we will now have a closer look.

The basic idea is to partition a fairly large register file into several partially overlapping register windows that accommodate the contexts in which procedure calls must be executed. Depending on the particular implementation, there are from 3 up to 32 such windows available of which, however, only one window is accessible at a time.

Each window includes 32 registers $r0 \ldots r31$, of which $r0 \ldots r7$ are global registers whose contents are the same in every window, and $r8 \ldots r31$ are 24 window registers whose contents are specific to procedure calls.

The window registers, in turn, are partitioned into sets of

- eight *outs* registers $r8 \ldots r15 \equiv o0 \ldots o7$ in which a procedure sets up the arguments for another procedure call;
- eight *locals* registers $r16 \ldots r23 \equiv l0 \ldots l7$ which accommodate local variable instantiations and temporaries;
- eight *ins* registers $r24 \ldots r31 \equiv i0 \ldots i7$ through which a called procedure receives its arguments from a calling procedure.

The *outs* registers of a calling procedure are in fact the same as the *ins* registers of the called procedure, i.e., the windows are overlapping, as schematically depicted in Fig. 14.6.

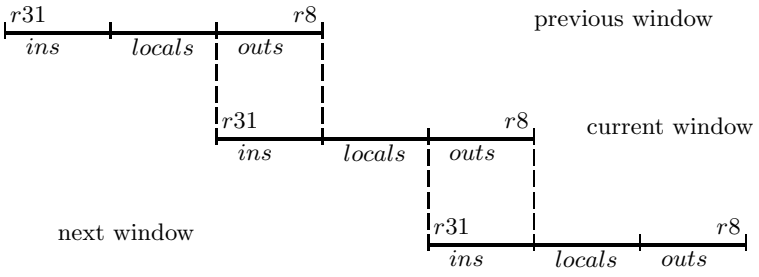


Fig. 14.6. Three consecutive register windows

Here we see a current window allocated to a procedure that executes, i.e., this window is the only one that is visible. It shares its *ins* registers with the *outs* registers of the previous window allocated to its calling procedure. The *outs* of the current window, in turn, are shared with the *ins* of a next window

that is going to be allocated to another procedure call. Thus the allocation of windows to procedure calls proceeds from left to right, and deallocation upon returning from procedure calls proceeds in the opposite direction.

With $n \geq 3$ windows in the register file, the number of registers that physically exist amounts to $n * 16 + 8$ window registers and 8 global registers, or $(n+1) * 16$ registers in total. However, since only one window is accessible at any time, it takes only 5 bits to address a register, irrespective of the actual number of windows, which is an important property for instruction formatting.[7]

With this window concept, there are two kinds of limitations that need to be dealt with.

The first one concerns the number of *ins*, *outs* and *locals* per window. Though the number of parameters of procedure calls rarely exceeds four and almost never six, provisions must nevertheless be made to cope with more parameters. The problem is a little more severe with the locals. Together with temporaries, there could be easily more than eight, of which some could, of course, be put into *ins* registers that are not used by the calling procedure, but local parameters in excess of this must inevitably spill over into memory. However, this concept may also lead to a considerable waste of register capacity in the case of procedure with only one or two parameters and about as many local variables.

At a higher level, the same problem comes up in the allocation of limited numbers of windows. Profiling a wide variety of conventional programs has revealed that on average there are about three and rarely more than eight nestings of procedure calls to handle, but with the advent of function-based languages this picture may drastically change in favor of a highly recursive programming style and, in consequence, far deeper nestings of procedure calls.

So, the machine has to provide a mechanism that in the case of a window overflow swaps a certain number of windows out into memory, and in the case of a window underflow swaps windows in from memory, provided there are windows left there. To this end, the machine supports a backup stack in memory. This stack associates with each register window a generously sized backup frame into which it may spill over whenever the computation runs out of available windows or registers within the windows.

The existence of this backup stack necessitates a stack pointer *sp* which is held in register $o6 \mid r14$. The stack pointer of the calling procedure becomes the (backup) frame pointer $fp = i6 \mid r30$ of the called procedure, relative to which all frame entries may be accessed with fixed offsets. Another register needs to be reserved for the return addresses of procedure calls, which is the *outs* register $o7 \mid r15$ of the calling procedure and, accordingly, the *ins* register $i7 \mid r31$ of the called procedure.

[7] The position of the visible window is held in a separate register that is not directly accessible by the program code.

It should be noted that there is no way of supporting nested procedure definitions. They would require static links to windows deeper down in the register file which would have to be made accessible. In fact, the origins of the SPARC architecture may easily be traced back to the programming language C. Except for globally free variables (which may be placed into the registers $g0 \ldots g7$), this language is flat in the sense that it knows only closed procedures (somewhat misleadingly called functions) whose contexts (or environments) can be put into single windows and, whatever does not fit in there, into the associated backup stack frames. The windows (and the stack frames) are completely unrelated to each other, except for the order in which they come into being, and in this sense fully conform to what has been said in Sect. 13.7 about the runtime environment for flat languages.

Control over the allocation and deallocation of windows and over window swapping is exercised by means of four count variables held in dedicated registers, which may assume values from the interval $0 \ldots n-1$. These are

- *cwp* (the current window pointer) which identifies the window position that is active (or visible);
- *cansave* which keeps track of the number of windows that can still be allocated;
- *canrestore* which denotes the number of windows that are occupied and may be released again;
- *otherwin* which gives the number of windows reserved for other uses, e.g., for trap handling in cases of window overflow or underflow.

Since the windows are counted cyclically $modulo(n)$, one window position must remain unused as its *ins* and *outs* registers overlap with usable windows. With this in mind, the above count variables must satisfy the equation

$$cansave + canrestore + otherwin + cwp = n - 1 \quad .$$

Allocating a new window is possible if $cansave > 0$, in which case the sequence of operations

$$cwp \leftarrow cwp + 1;\ cansave \leftarrow cansave - 1;\ canrestore \leftarrow canrestore + 1;$$

must be executed to adjust the count variables, otherwise a trap is generated. The trap calls a kernel routine that swaps the contents of older windows out into the backup stack, thus making new windows available for allocation. Likewise, releasing an existing window is possible if $canrestore > 0$, in which case the count variables must be updated by the complementary sequence

$$cwp \leftarrow cwp - 1;\ cansave \leftarrow cansave + 1;\ canrestore \leftarrow canrestore - 1;$$

otherwise a kernel routine takes over to fill empty windows with frames from the backup stack to continue with further computations after returning from procedure calls.

14.2.2 Some Important SPARC Instructions

Though we are not really interested in bits and bytes here, a few words on instruction formats should nevertheless be said since they more or less dictate how the instructions need to be sliced and what they can do. Simplicity of interpretation, which is what RISC architectures are all about, requires that all instructions have fixed word size (say, of 32 bits). They can then be fetched from (cache) memory and loaded into an instruction register for decoding in one machine cycle and in one piece. Moreover, there should be only a few formats in which instruction words are subdivided into dedicated operator and operand (register) address fields. Given these constraints, there is little room left to choose among many alternatives.

In fact, the SPARC architecture distinguishes just three classes of word-sized instruction formats, of which the most important one has the general form

$$\text{RATOR } sr1\ sr2\ dr\ ;\ .$$

Here RATOR, $sr1$, $sr2$, dr denote an operator, the addresses of a first and a second source register for operands, and the address of a destination register for results, respectively.[8] This format applies to several control instructions, to all value-transforming instructions (which are exclusively register to register) and, in slightly modified form, also to load | store instructions that transfer data between registers and memory.

The control instructions that effect branches to and returns from procedure include the following:

CALL $dest$ saves in register $o7$ | $r15$ the current pc as the return address and then branches to procedure code at the address $pc + off$, where $off = (dest - pc)/4 \in [\, -2^{29} \ldots 2^{29} - 1\,]$.[9]

JMPL $sr1\ sr2\ dr$ branches to procedure code in that it saves the current pc as the return address in the destination register dr, and then computes a new pc by adding the contents of the source registers $sr1$ and $sr2$. Alternatively, the source register $sr2$ may be replaced with an immediate value specifying an offset that is added to the contents of $sr1$; other than that, the instruction has the same effect. This instruction is also used in degenerate form to return from a procedure call.

RETURN $sr1\ sr2$ returns from a procedure by deallocating a window, as described in the preceding subsection, and then setting the pc to the value obtained by adding the contents of the two source registers $sr1$ and $sr2$ in the window to which control returns.

[8] Wherever it makes sense, the second source register may be replaced with an immediate value specifying, say, an offset relative to a base address held in the first source register.

[9] This instruction is in a format class of its own that features a 2 bit operator code followed by the 30 bit offset off.

RET without any parameters has the same effect as JMPL $i7$ #0 $g0$, i.e., it returns control to the address held in register $i7$ without deallocating the current window.

SAVE $sr1$ $sr2$ dr allocates, as described in the preceding subsection, the next window of the register file and then overwrites the destination register dr of the new (the callee's) window with the sum of the contents of the source registers $sr1$ and $sr2$ (or alternatively an offset) of the caller's window.

RESTORE $sr1$ $sr2$ dr switches back to the caller's window, as described in the preceding section, and then updates the destination register dr in it with the sum of the contents of registers $sr1$ and $sr2$ (or alternatively an offset) in the callee's window.

REST without any parameters has the same effect as RESTORE, i.e., it returns to the caller's window, except that it does not perform the subsequent addition.

It may be noted that, for good reasons, the instructions that call and return from procedures may be separated from the instructions that switch windows. This is to avoid unnecessary switches whenever a procedure call is the last one in a sequence of nested calls.

There is, of course, a conditional branch instruction that, as in the CISC machine of Sect. 14.1, may be spezialized by a condition code cc:

BR_cc $dest$ sets the program counter pc to $pc + off$ (where $off = (dest - pc)/4 \in [\, -2^{21} \ldots 2^{21} - 1 \,]$) if the condition specified by cc becomes true, otherwise code execution continues at $pc+4$, i.e., with the next instruction in sequence. The branch instruction belongs to yet another format class which provides a field for a 22 bit signed offset.[10]

There are basically three move instructions available to copy data from one place to another:

ST.F $sr1$ $sr2$ dr stores in the format .F the contents of register dr in memory at the address obtained by adding the contents of $sr1$ and $sr2$.

LD.F $sr1$ $sr2$ dr loads in the format .F into register dr the contents of the memory address obtained by adding the contents of $sr1$ and $sr2$.

MOVE.F $cond$ $sr2$ dr copies in the format .F the contents of register $sr2$ to register dr if the condition $cond$ is true; otherwise nothing happens. If the condition is missing, the move is done unconditionally.

Representative of the instructions that do the actual computing are

ADD.F $sr1$ $sr2$ dr which adds numbers in the format .F in the source registers $sr1$ and $sr2$, and puts the result in register dr.

CMP.F $sr1$ $sr2$ which compares in the format .F the contents of $sr1$ and $sr2$, and accordingly sets condition codes to be inspected by subsequent conditional branch instructions.

[10] In another instruction SETHI that belongs to the same format this field is used for immediate operands.

The last instruction that needs to be mentioned is NOP (short for no operation) which has no effect other than killing time, exactly one machine cycle to be precise. This instruction occurs quite frequently in RISC code as a consequence of pipelined instruction processing. Following conditional branches taken, the pipeline has to be restarted with new instructions beginning at the branch addresses while instructions already in the pipeline may have to be discarded. The NOPs are used simply to fill the ensuing slots that open up behind the branch instructions, unless they can be filled with useful instructions by code reorganization.

14.2.3 The SPARC Assembler Code for Factorial

To illustrate how SPARC code typically looks like, we consider what may be obtained by compiling the function definition

$$fac\, n \;=\; \texttt{if}\ (\ n \geq 1\ )\ \texttt{then}\ (\ n * (\ fac\ (-\ 1\ n\ )\ )\ \texttt{else}\ 1\ \ .$$

Figure 14.7 shows the code generated by a compiler for an equicalent C program, with all optimizations turned off.

The first thing that is done when this code is entered at label fac is to have the instruction SAVE allocate a new window in the register file and also a new frame of a generously chosen 120 bytes in the backup stack, simply by advancing the stack pointer sp accordingly. The single parameter n received through the ins register $i0$ is stored in the backup frame entry $fp + 68$ and from there is immediately reloaded into the $outs$ register $o0$. The comparision performed by the instruction CMP realizes the inverse of $(n \geq 1)$, i.e., the subsequent branch by the instruction BR_le is taken if the value of n drops below 2. However, since the branch would become effective in the pipeline two instructions later, the compiler routinely fills the slot in between with the instruction NOP, as it does following every other branch instruction in the code.

The following three instructions compute the expression $(n-1)$ and place the result into the $outs$ register $o0$, where it is expected as input parameter by the subsequent recursive call of fac. The function value that is returned through the ins register $i0$ of the callee and thus becomes the $outs$ register $o0$ of the caller is, in preparation for the multiplication $(n * (fac\ (-\ 1\ n))$, moved to register $o1$ to make room for reloading the actual value of n from the frame entry $fp + 68$ to register $o0$.

The multiplication is performed by a library routine *umul* which implements it as a sequence of add and shift instructions compatible with pipeline processing of one instruction per machine cycle.[11] This routine expects its parameters in the caller's $outs$ registers $o0$ and $o1$ and returns its result in

[11] The more advanced SPARC III processor supports MULT instructions in conjunction with a hardwired multiplier that does multiplication in one machine cycle.

```
fac : SAVE sp #-120 sp   /* entry into fac code, allocates
                            new window and backup stack frame*/

      ST.W fp #68 i0     /*passes the the function parameter n
      LD.W fp #68 o0     from i0 to o0 */

      CMP o0 #1          /*computes n ≤ 1 and
      BR_le ll3          branches to label ll3 if true*/

      NOP                /* fills empty slot after a branch*/

      LD.W fp #68 o0     /*loads parameter n again */
      ADD.W o0 #-1 o1    /*computes n - 1 */
      MOVE.W o1 o0       /* sets up n - 1 in o0 for next
                            call of fac*/

      CALL fac           /*recursively calls fac again*/
      NOP

      MOVE.W o0 o1       /*moves (fac (- 1 n)) to o1 */

      LD.W fp #68 o0     /*loads n again*/

      CALL umul          /*calls multiplication routine*/
      NOP

      ST.W fp #-20 o0    /*stores result of multiplication */

      BR.A ll4           /*branches unconditionally to label ll4*/
      NOP

ll3 : MOVE.W #1 o0       /*copies value 1 of else clause
      ST.W fp #-20 o0    to o0 and to frame entry fp - 20*/

ll4 : LD.W fp #-20 i0    /*loads function value*/
      BR.A ll2
      NOP
ll2 : RET                /*returns from call of fac and
      RESTORE sp #120 sp deallocates current window and stack frame*/
```

Fig. 14.7. Nonoptimized SPARC code for the factorial

register $o0$ of the caller's window. The instruction ST.W moves this result to a backup stack entry at the address $fp - 20$ outside (below) the current frame.

Continuing at label $ll4$, the function value is reloaded again from this stack entry into the *ins* register $i0$ through which it is returned to the callers *outs*

register $o0$ once control has returned to it and the instruction RESTORE has the caller's window restored.

Alternatively, if the conditional branch to label $ll3$ is taken by the instruction BR_le, the constant value 1 is written into the $outs$ register $o0$ and into stack entry $fp - 20$ to be returned as the value of the function call that terminates the recursion.

Though this piece of code closely reflects the structure of the high-level program, it also shows how poorly both the window registers and the frames of the backup stack are utilized if there is just one formal parameter to be dealt with. Taking into account the overlap between $outs$ and ins of two successive (a caller's and a callee's) windows, the entire computation in this particular case uses just the two $outs$ registers $o0$ and $o1$, with $o0$ serving as the register through which argument values are passed from callers to callees and function values are handed over in the opposite directions, i.e., not counting the globals, 14 out of 16 window registers remain unused. Similarly, only one out of 30 word-sized entries is used in each of the backup frames to pass the argument value along. Another location underneath the topmost stack frame is used to return function values.

A decidedly better register utilization can be achieved by having the code optimizer inline (or flatten) several consecutive recursive calls of fac. Each instance of the function body code must then be distinguished by the use of distinct registers. If two registers are required for each instance, and the $locals$ registers are engaged in this game as well, it is then possible to inline up to eight calls. As inlining eliminates time-consuming call/save and return/restore sequences and generally opens up more opportunities for other optimizations, the resulting code not only utilizes registers more economically but also runs a lot faster. However, the alienation from the original high-level program may be quite considerable, particularly as programs become more complex and inlining can be applied to (nestings of) different procedure calls.

Other redundancies can be found in the factorial code itself. They are primarily due to repeated reloads of registers from the corresponding backup stack entries. A code optimizer would replace the first occurrence of the instruction LD.W fp #68 $o0$ with MOVE.W $i0$ $o0$, and drop the second occurrence altogether since it would find out that the value of n is still in $o0$. Likewise, the instruction LD.W fp #-20 $i0$ at label $ll4$ could be replaced with MOVE.W $o0$ $i0$ since the right return value is in $o0$ irrespective of how this instruction is reached. A small amount of code reorganization would move at least in the place of the first occurrence of the NOP instruction either the LD.W or the ST.W instruction that now precedes the conditional branch instruction BR_le, thus saving another pipeline slot for useful computations.

14.3 Summary

Having had a closer look at two representative architectures of real computing machines, we note that differences from the various code-executing abstract machines described in the preceding chapters, particularly the imperative abstract machines of Chap. 13, primarily concern implementation issues. All these machines, of necessity, share essential control instructions to call and return from procedures and to branch conditionally, instructions that move things from one place to another, and instructions that do the actual computing, i.e., arithmetic, logic and relational operations.

However, whereas the instructions of the abstract machines operate on (pointers to) conceptually infinite data structures such as stacks and linked lists of things held in heaps, finiteness of resources and bandwidth limitations are a major concern when it comes to implementing and operating on such structures in real machines. Fixed sizes of registers and addressable memory entries dictate data and instruction formats. Data formats determine the instructions needed to operate on data, instruction formats limit the total number of instructions, the number of operands, and the addressing modes that can be supported. And finally, the addressing modes and address formats determine the address ranges that may be covered.

A particularly challenging problem is the utilization of limited numbers of processor-internal registers that replace the working (value) stacks of our abstract machines. Runtime efficiency demands that all data and (memory) addresses that belong to the current focus of computation be held in these registers, which requires carefully tuned allocation strategies. Larger register sets alleviate this allocation problem to some extent. However, they do so at the expense of logarithmically growing register addresses which may be difficult to squeeze into given instruction formats, and also of increased context-switching times which may to some extent offset the performance gains made by adding registers.

Another problem with registers is that they can be addressed only directly. There is no way of keeping register addresses in registers and using them for indirect addressing. This precludes building structures such as linked lists across register contents, with the consequence that runtime structures composed of statically linked frames cannot be held even in sufficiently large register files but must be kept in memory. What is, however, possible are register-resident frames holding instantiations of closed procedures that, other than being lined up in their order of creation, are totally unrelated to each other.

The chapter has described both a CISC (complex instruction set computer) architecture that is capable of supporting in memory a runtime environment with statically linked activation records for invocations of nested procedures, and a *RISC* (reduced instruction set computer) architecture that works with register windows for invocations of flat (or closed) procedures.

The CISC instruction set is fairly high-level, closely resembling that of the MC680x0 family, which facilitates compilation of high-level language programs

into dense code. The instructions may have varying lengths as operators may be paired with a variety of addressing modes for operands that are tailored to the needs of pushing, popping and addressing, also over levels of indirection, stack entries held in memory. To keep the code relocatable, branching to procedures or to the alternatives of conditionals is done with fixed offsets relative to the program counter. Instruction decoding requires sophisticated controls and, depending on address positions, addressing modes used for operands and operand formats, it may take widely varying numbers of machine cycles to execute.

The CISC register set is fairly small, with most of the action taking place in the memory-resident runtime stack. The registers merely accommodate the temporaries and the base addresses of data structures – most prominently the runtime stack – that constitute the current scope of the computation. Since there are eight registers available for address manipulations, this architecture is a perfect vehicle for the direct implementation of our code-executing abstract machines that feature several stacks and a heap.

The SPARC family that we have chosen as a typical RISC architecture centers around register files of sizes that may vary in multiples of 16 registers. However, only one window of 32 registers is addressable at any time. It comprises 8 registers for global values and 24 registers that accommodate the incoming, local and outgoing parameters, i.e., basically the activation record, of a procedure call. Moving the accessible window in a round robin fashion across the register file in increments (or decrements) of 16 registers mimics the workings of a runtime stack as the *ins* registers of some current window overlap with the *outs* of the preceding window to pass parameters from a calling to the called procedure. Other than that, there are no linkages between the windows, meaning that only closed procedures that contain no references to items held in other windows may be supported. This concept clearly has its origins in the programming language C. This language knows only flat procedure definitions that, except for references to global variables, are closed and thus are perfect candidates for compilation to SPARC code. In fact, C is about the only language for which compilers to SPARC code exist. All other high-level languages are compiled to C as an intermediate language to take advantage of the C compiler backend for the generation of highly efficient machine code.

Since there are only finitely many window positions (from three to eight, depending on the processor implementation chosen), a backup stack must be supported in memory, into which older windows may spill over to make room in the register file for more procedure calls, and from which empty window positions may be reloaded when returning from procedure calls. The backup stack may also accommodate data items in excess of the fixed window capacity.

The SPARC instruction set is very much in line with that of other RISC architectures, meaning that they are tailored for smooth and efficient pipeline execution. All instructions are simple enough to pass through each pipeline stage in one machine cycle, most of them perform register-to-register opera-

tions, there are explicit load and store instructions to move things in and out of memory, and branch instructions, as usual, work with offsets relative to the program counter. There are two instructions unique to SPARC that control window movement: SAVE advances the window position from caller to callee and at the same time allocates another frame on the backup stack, RESTORE returns to the preceding window position and deallocates the current backup frame upon returning from callee to caller. Both instructions effect traps in cases of window overflow or underflow to move windows out to or in from the backup stack, respectively.

References

Literature on conventional computer architecture and machine level programming is abundantly available in the form of textbooks. Of immediate interest with regard to the contents of this chapter are, for instance, the texts by Harman and Hein on the MC68000 processor family [HaHe95], by Antonakos on the MC68000 and on the Intel 80x86 families [Ant98, Ant99], by Livadas and Ward on computer organization exemplified by the MC68000 [LiWa93], all of which address both architecture and assembler programming. Texts on the SPARC architecture are the manual by Weaver [Wea93], the book by Paul [Pau99], and material made available by SUN. More on RISC processors may be found in the textbooks by Patterson and Hennessy [PaHe90], which includes a thorough description of the MIPS architecture and of pipelined instruction execution, and the texts by Levine and by Weiss and Smith on the PowerPC [Lev94, WeSm94]. Another more recent book by Bryant and O'Hallaron [BrHa03] looks at computer architecture from a machine-level (assembler) programming perspective, addressing the implementation of conditionals, loops, procedure calls and parameter passing mechanisms. These issues are also extensively covered in textbooks on compilers, e.g., the one by Appel on compiler implementation in C [App99] which also contains a chapter on compiling functional languages.

A

Input/Output

Throughout the discussion of the various abstract machines there has been something important missing: an answer to the question of how a program expression or executable program code gets into the machine, say from a user interface (typically a keyboard/display arrangement) or some file held in a peripheral store, and how the result of a program run is returned to that interface or written back into a file again. In a more general setting, we might wish to know how a running program can be made to repeatedly communicate with a user, or another computation for that matter, by requesting (or receiving) input of some kind and responding with appropriate output.

At first sight, we could simply have the input/output problem taken care of by the traditional mechanisms known from conventional computing systems, and the issue would be settled. Unfortunately, things are not that simple in the world of λ-calculus-based languages of the AL variety.

To understand what is at stake here, we need to realize that input/output always involves interactions with the state of an environment consisting of files as abstractions of a wide variety of devices that hold data. Getting input from a file means copying all or part of it, putting output into a file means overwriting all or part of it, which is to say that input/output operations are performed for their effects on the environment. Since this is exactly what assignment statements of imperative languages are doing as well, just at a finer level of granularity and on specific components of the runtime environment (the stack and the heap) (see Chaps. 13 and 14), input/output is fully compatible with the imperative model of computing. This model imposes a sequential execution order upon all updating and copying operations on the state, whether assignments or input/output, which trivially guarantees determinacy of results for all program runs in the same environment.

This contrasts with the λ-calculus where the Church–Rosser property guarantees the determinacy of results (normal forms) irrespective of the order in which redices that are only partially ordered are reduced, the reason being that evaluating a λ-expression is based on context-free substitutions of equals by equals without interaction with a state.

This situation changes, of course, if input/output operations are introduced into the λ-calculus in naive form. To illustrate what may happen, consider a simple machine that, in addition to the structures that are necessary to perform reductions, also includes two abstract registers *in* and *out* through which expressions may be communicated with an environment. Both registers may either be filled with valid expressions or be empty. We also assume two primitive functions `get` and `put` which read from the *in* register and write to the *out* register, respectively. The primitive `get` implicitly takes as an argument the contents of the *in* register and reproduces it as a function value, whereas `put` takes as an argument the expression to be moved out and returns as a value some dummy symbol, say $\oslash$, in its place. The primitive `get` is executable only if the *in* register is filled, consuming its contents and leaving it empty, and `put` is executable only if the *out* register is empty, filling it with the argument, otherwise the applications remain unchanged in both cases.

Here we have a very simple program that, depending on its interleaving with executable `get`s and `put`s, does weird things with its output:

$$\begin{aligned} \texttt{letrec}\ f &= \texttt{lambda}\ u\ v\ \texttt{in}\ (g\ v\ u) \\ g &= \texttt{lambda}\ u\ v\ \texttt{in}\ (h\ u\ u) \\ h &= \texttt{lambda}\ u\ v\ \texttt{in}\ \texttt{"nil"} \\ \texttt{in}\ (f\ (\texttt{put}\ \texttt{"hello"})\ (\texttt{put}\ \texttt{get}))\ &. \end{aligned}$$

Assuming that the register *in* is going to be filled with the string "`world`", we may get the following output sequences:

- "`world`" "`hello`" if both `put` applications can be reduced before applying f;
- "`hello`" "`world`" if both `put`s are passed along by f as they are and reduced before applying g;
- "`hello`" "`hello`" if the (`put get`) application is passed along by g and reduced before applying h (the other `put` disappears);
- and no output at all if the (`put get`) application falls through the application of h, which returns "`nil`" as its value.

The problem that we face here is that interactions with an environment as naive add-ons to the λ-calculus obviously violate the Church–Rosser property. They produce results – output in this particular case – that are dependent on execution orders and thus are incompatible with computations based on a reduction semantics (or context-free substitutions).

There are several ways of getting around this problem, of which the more important ones will be outlined in the following sections.

A.1 Functions as Input/Output Mappings

In its simplest form, an AL program may interact with the environment by receiving a single argument object, say a string of characters, via a `get_s`

operation and returns to the environment a result which is another character string, using a `put_s` operation. This may be expressed as an application of a function *main* to the input operation `get_s`, which in turn becomes the argument of an application of the output operation `put_s`:[1]

$$(\texttt{put\_s}\ (\mathit{main}\ \texttt{get\_s}))\ .$$

Unlike the `put` and `get` operations of the naive approach, executing `get_s` and `put_s` must include a synchronization mechanism, signified by the postfix `_s`, which ensures that *main* is not applied before input can be read from the *in* register, and output cannot be produced before the evaluation of the *main* function's body has terminated. Otherwise, the `get_s` may be duplicated in the body of *main*, of which each copy attempts to retrieve another entry from the *in* register, or the `put_s` may produce output that still contains occurrences of `get_s`.

To formalize the specification of this and the following input/output schemes, we use the general notation

$$C[\ e\ ]\ \|\ <\mathit{in}\ \mathit{out}> \rightarrow C[\ e'\ ]\ \|\ <\mathit{in}'\ \mathit{out}'>\ \ |\ e \mapsto e'\ .$$

It describes the transformation, by a sequence of (β-)reductions denoted by the arrow $\mapsto$, of some expression e that may (or may not) be embedded in some context C, e.g., a larger expression surrounding it, and an associated environment (or state), represented by its *in* and *out* register interface, into an expression e' and a new environment.

Evaluation of the above application may be specified by the following three rules:[2]

$$C[\ (\mathit{main}\ \texttt{get\_s})\ ]\ \|\ <\mathit{string}\ \mathit{out}> \rightarrow C[\ (\mathit{main}\ \mathit{string})\ ]\ \|\ <\Box\ \mathit{out}>\ ,$$

$$(\texttt{put\_s}\ \mathit{string})\ \|\ <\mathit{in}\ \ \Box> \rightarrow \oslash\ \|\ <\mathit{in}\ \ \mathit{string}>\ ,$$

$$C[\ e\ ]\ \|\ <\mathit{in}\ \mathit{out}> \rightarrow C[\ e'\ ]\ \|\ <\mathit{in}\ \mathit{out}>\ \ |\ e \mapsto e'\ ,$$

where $C[\ e\ ]$ denotes all contexts of e.

This restricted form of interaction with the environment is in compliance with the notion of functions as mappings from inputs to outputs. It never interferes with the evaluation of the body of *main*, assuming that this body does not contain further `get_s`s and `put_s`s. What we have here is in fact equivalent, on a larger scale, to an assignment statement: the function *main* may be seen as the expression on its right-hand side, the `get_s` operation is equivalent to copying values for the expression's variables from the store, and the `put_s` operation does the equivalent of assigning the value of the expression

[1] Multiple arguments and result values may be packed in lists to comply with the single-argument/single-function-value notation.

[2] The symbol $\Box$ denotes an empty input or output register.

to the variable on the left (which are the variables *in* and *out*, respectively, in this input/output scheme).

There are, however, limitations to this approach: since the strictness assumption demands that the input be fully specified and that output must not be produced before the computation terminates, there is no way of handling, say, a continuous flow of input that generates a continuous flow of output, which would establish a simple form of repeated interaction with the environment.

Such interactions may be realized by means of streams, i.e., potentially unending sequences of objects that can be exchanged between a program and its environment. These sequences may be thought of as infinite lists that require nonstrict evaluation, meaning that elements may be taken off their front ends and appended to their back ends without having the lists themselves evaluated.

To illustrate the use of streams, consider a simple request/response interaction scheme that works with a first stream of requests to be communicated to the environment and a second stream to accumulate responses received from the environment. Requests are assumed to be of the form

$$request =_s (\texttt{putstring}\ string) \mid \texttt{getstring}\ ,$$

i.e., they may either put a string value into the *out* register or receive a string value from the *in* register. Responses have the form

$$response =_s < \texttt{success}\ string > \ \mid \ < \texttt{failure}\ string > \ ,$$

signifying success or failure of an interaction, with *string* denoting either the interaction's return value or the cause of the failure.

A function *main* that communicates with an environment by means of request/response streams may be specifies as

$$main = \texttt{lambda}\ u\ \texttt{in}\ e_{main}\ ,$$

where u denotes a formal parameter to be substituted by a special variable *resp* for response streams that are initially empty.

Executing an application of this function to *resp* may be governed by the following rules of an input/output scheme:

$$C[\,(main\ resp)\,] \parallel < in\ \ out > \rightarrow C[\,e_{fun}[\,u \leftarrow resp\,]\,] \parallel < in\ out > \ ,$$

$$C[\,e\,] \parallel \ < in\ out > \rightarrow C[\,e'\,] \parallel < in\ out > \ \mid \ e \mapsto e' \ ,$$

$$\texttt{getstring} : e \parallel < string\ out > \rightarrow e[\,resp \leftarrow < \texttt{success}\ string > : resp\,] \parallel < \Box\ \ out > \ ,$$

$$\texttt{getstring} : e \parallel < \texttt{eos}\ out > \rightarrow e[\,resp \leftarrow < \texttt{failure}\ \text{"}\texttt{stream\_end}\text{"} > : resp\,] \parallel < \Box\ out > \ ,$$

$$(\texttt{putstring}\ string) : e \parallel < in \ \square > \rightarrow$$
$$e[\ resp \leftarrow < \texttt{success}\ \text{"}\texttt{putstring}\text{"} >: resp\ ] \parallel < in\ string > \ ,$$

$$<> \parallel < in \ \square > \rightarrow \texttt{done} \parallel < in\ \texttt{eos} > \quad .$$

It may be noted that the first rule is just a special case of the second rule. It has nevertheless been included to make explicit the substitution of the response stream into the body of *main*.

The rules for `getstring` either take a valid string from the *in* register and prepend a $<$ `success` *string* $>$ tuple to the response stream or, if the special end-of-stream symbol `eos` is encountered, signify failure. The rule for `putstring` puts its argument string into the *out* register and prepends a tuple signifying successful completion of the operation to the response stream. It is important to note that prepending something to the response stream is seen by all occurrences of *resp* since all of them are substituted by the new sequence.

Otherwise, we have a rule for normalization of an expression without engaging input/output, and a rule for termination of the output if the request stream is empty.

A primitive login procedure that just asks the user for her/his name and responds with "hello" and the name that is being returned would have to look like this:

$$main = \texttt{lambda}\ u\ \texttt{in}$$
$$(\texttt{putstring}\ \text{"}\texttt{your name ?}\text{"}) : \texttt{getstring} : (hello\ (\texttt{rest}\ u))$$

$$hello = \texttt{lambda}\ v\ \texttt{in}$$
$$(\texttt{case}$$
$$< \texttt{success}\ name >: tail \rightarrow$$
$$(\texttt{putstring}\ \text{"}\texttt{hello}\text{"}) : (\texttt{putstring}\ name)$$
$$\texttt{otherwise}\ldots$$
$$v) \quad .$$

The body of the function *main* is a stream of two explicit requests followed by a call for the subfunction *hello*. This function performs a pattern match on the response stream, expecting a `success` message as an echo to the `getstring` of the request stream, and thereupon produces the desired answer as another out`put` request (reactions to mismatches are ignored here).

Correct evaluation of such stream programs requires that the `putstring` and `getstring` operations are atomic in the sense that they are completed only after responses have arrived and been prepended to the response stream, i.e., output and input are tightly synchronized. Synchronization problems of another kind may nevertheless occur if, owing to programming errors, unsuccessful attempts are made to inspect responses before the corresponding requests have been made, in which case the program would deadlock. This

would, for instance, be the case if the calls of *hello* and `getstring` would be interchanged in the request stream generated by the above function *main*.

Another way to have a program communicate with an environment that is based on the notion of functions as mappings from input to output is called environment passing. The idea is that functions are applied to complete current environments and return new, modified environments as function values. These environments are also referred to as world states, or worlds for short.

In its simplest form, and using the same notation as before, such interactions could be specified by three transformation rules that take the *in* and *out* registers as representations of the environment (or of the world state):

$$C[\ (main\ \texttt{getstate})\ ]\ \|\ <in\ out> \rightarrow \\ C[\ (main\ <in\ out>)\ ]\ \|\ <\Box\ \Box>\ ,$$

$$(\texttt{putstate}\ <in\ out>)\ \|\ <\Box\ \Box> \rightarrow \texttt{done}\ \|\ <in\ out>\ ,$$

$$C[\ e\ ]\ \|\ <\Box\ \Box> \rightarrow C[\ e'\ ]\ \|\ <\Box\ \Box>\ \mid e \mapsto e'\ .$$

The first rule uses the primitive `getstate` to move the entire state into the argument position of the function *main*, leaving the original state empty. The second rule applies `putstate` to copy back as the new world state the $<$ *in out* $>$ tuple that has emerged as the value of the *main* application. In between, normalization of an expression may proceed without interactions with the state (the last rule).

For this to work correctly, the entire interaction would have to be specified as a nested application

$$(\texttt{putstate}\ (main\ \texttt{getstate}))\ .$$

There must be no `putstate`s or `getstate`s in the body of *main*, which means that interaction with the state takes place only when starting and termintating *main*.

However, this is not really a satisfactory solution. The more ambitious objective of having a functional program repeatedly interact in an orderly way with the environment, or the world state, between start and finish is more difficult to accomplish. The problem is that there must be a tight synchronization between an internal representation of the state (the one used on the left of the separator $\|$) and the real state of the environment (on the right of $\|$), which may be changed by operations external to the program under consideration. That is to say, every input operation must find the states on both sides of the separator $\|$ to be the same, and every output operation must effect the same changes on both sides of the $\|$ symbol. To put it another way, there must be exactly one representation of the world state that must be the same as the real state, i.e., the representation must be unique.

Irrespective of concrete realizations, this uniqueness property may be expressed by the following transformation rules that describe interactions with

the world state with the notion of a unique world context UWC in which input/output operations take place. This specific context, whose existence can be asserted by program analysis such as uniqueness type checking, essentially ensures that an input/output operation is in exclusive possession of the world state when it is executed.[3]

$$\texttt{start}\ main \parallel < in\ out > \rightarrow (main\ < in\ out >) \parallel < in\ out >\ ,$$

$$UWC[\ (\texttt{Getstring}\ < string\ out >)\ ] \parallel < string\ out > \rightarrow UWC[\ << \texttt{success}\ string > < \Box\ out >>\ ] \parallel < \Box\ out >\ ,$$

$$UWC[\ (\texttt{Getstring}\ < \texttt{eos}\ out >)\ ] \parallel < \texttt{eos}\ out > \rightarrow UWC[\ << \texttt{failure}\ \text{"}\texttt{stream\_end}\text{"} > < \Box\ out >>\ ] \parallel < \Box\ out >\ ,$$

$$UWC[\ (\texttt{Putstring}\ string\ < in\ \Box >)\ ] \parallel < in\ \Box > \rightarrow UWC[\ << \texttt{success}\ \text{"}\texttt{Putstring}\text{"} > < in\ string >>\ ] \parallel < in\ string >\ ,$$

$$C[\ e\ ] \parallel < in\ out > \rightarrow C[\ e'\ ] \parallel < in\ out >\ \mid e \mapsto e'\ ,$$

$$< in\ \Box >\ \parallel < in\ \Box > \rightarrow \texttt{done} \parallel < in\ \texttt{eos} >\ .$$

Here is, as an example that uses this uniqueness-based environment-passing mechanism, a function *main* that realizes the primitive login procedure described above:

```
main = lambda u in
    letrec f = lambda v w in
      ( case
          < "getstate" < in out > > ‖ (eq w "w0") →
                      ( f ( Putstring "your name ?" < in □ > ) "w1" )
           << success "Putstring" > < in "your name ?" >> ‖
                  (eq w "w1") → ( f ( Getstring < name out > ) "w2")
          << success name > < □ out >> ‖ (eq w "w2") →
                     ( f ( Putstring "hello" : name < in □ > ) "w3")
          << success "Putstring" > < in "hello" : name >> ‖
                                                  (eq w "w3") → ...
        otherwise ...
                v )
          in ( f < "getstate" v > "w0" )
```

The body of *main* defines a recursive function f that moves through a sequence of pattern matches that produces the desired communication with the state. The correct sequencing of the `Putstring`s and `Getstring`s has

[3] The first of these rules applies a `start` operator to some function *main* that simply copies the world-state representation in its argument position.

been accomplished in this program by means of a little trick: the function f is equipped with another parameter w that may assume the values "$w0$", "$w1$", "$w2$", ... which change and are subsequently tested whenever f is called. This trick gets us around the problem that the sequencing should in reality be determined by data dependencies that are part of the world states. However, since we have chosen to represent the world state just by *in* and *out* registers through which strings are passed, the additional parameter w simply assumes the role of a full world representation, as far as sequencing is concerned.

A major deficiency of both styles of interaction – streams and environment passing – is the low level of abstraction. The details of sequencing interactions with the environment must be explicitly specified, either by means of streams or through dependencies among world states and uniqueness properties. In this regard, the programming style does not differ much from that of imperative programming. However, an expression-oriented programming model should provide the means to introduce abstractions that hide the details of stream handling and environment passing.

A.2 Continuation-Style Input/Output

The basic idea here is again to focus on one interaction at a time and to specify how the computation must continue after the interaction has occurred. This result continuation is realized by means of a function of one parameter through which the result of the interaction is passed along. Its body may recursively contain further constructs of the same kind, possibly terminating with an empty result continuation, symbolized by `done`. Thus, a program that performs several interactions is composed of a nesting of an equal number of result continuations, of which each handles one interaction. More precisely, a program just specifies requests for interactions that are performed by the environment itself.

Using the notation (*cont_fun response*) for the application of a continuation function *cont_fun* to an interaction response named *response*, interaction by result continuation can be specified by the following rules:

$$(\texttt{GetString}\ cont\_fun) \parallel < string\ out > \rightarrow (cont\_fun\ < \texttt{success}\ string >) \parallel < \Box\ out > \ ,$$

$$(\texttt{GetString}\ cont\_fun) \parallel < \texttt{eos}\ out > \rightarrow (cont\_fun\ < \texttt{failure}\ \text{"}\texttt{stream\_end}\text{"} >) \parallel < \Box\ out > \ ,$$

$$(\texttt{PutString}\ string\ cont\_fun) \parallel < in\ \Box > \rightarrow (cont\_fun\ < \texttt{success}\ \text{"}\texttt{PutString}\text{"} >) \parallel < in\ string > \ ,$$

$$C[\, e \,] \parallel < in\ out > \rightarrow C[\, e' \,] \parallel < in\ out > \ \mid e \mapsto e' \ ,$$

$$\texttt{done} \parallel < in\ \Box > \rightarrow \texttt{done} \parallel < in\ \texttt{eos} > \ .$$

The first rule simply replaces the name *main* of its main function by the right-hand-side expression e of its defining equation $main = e$. The next three rules have applications of the primitives `GetString` and `PutString` to some continuation function *cont_fun* transformed into the application of *cont_fun* to the results of the respective interactions. The second to last rule evaluates expressions without interactions, and the last rule takes care of the termination condition.

In this result continuation style, our primitive login procedure can now be reformulated as

```
main = ((PutString "your name ?")
          lambda – in
            (GetString
                 lambda u in
                   (case
                      < success name >→
                          ((PutString < "hello" name >)
                                lambda – in done)
                       otherwise ...
                    u))) .
```

Here we have three nested applications of continuations corresponding to the three interactions – requesting a name through `PutString`, receiving it through `GetString` and responding with `"hello"` and the name – that need to be performed. The continuations are `lambda` abstractions, of which those whose parameters are denotes as '–' simply dump their interaction's responses (both `PutString`s), and the continuation following `GetString` includes a pattern match to extract the name from the response < `successor` *name* > and to pass it on to the subsequent `PutString` interaction.

At first glance, continuation-style programming seems to be all we need to include interactions with an environment in function-based programs. Compositions of primitive interactions can be had as part of the construction of result continuations. However, the tight coupling of interactions and continuations renders it impossible to separate the two from each other, i.e., to perform an interaction without explicitly specifying in place how the response ought to be dealt with immediately afterwards. Working around this problem would mean passing continuations around as parameters, say, when large programs need to be composed of smaller components, with the consequence that continuation parameters may be scattered all over the place.

A more elegant solution consists in a construct that makes explicit the composition of an interaction and a continuation function by means of a primitive

function `bind`:

$$(\texttt{bind}\ \mathit{interaction}\ \mathit{cont\_fun})\ .$$

This bind construct evaluates in two steps: first it evaluates its first argument, the interaction description (which includes performing the interaction), returning an intermediate expression (`return` *value*). It is itself an identity interaction description that simply returns *value*. The application of `bind` is, in the second step, evaluated only if its first parameter is such a `return` interaction, passing its return value *value* on to the continuation function $\mathit{cont\_fun}$.

The important difference from the result continuation is that this monadic style of interaction programming allows complex interactions to be specified as compositions whose evaluation terminates with a primitive interaction that results in a `return` application.

The transformation rules for these monadic interactions, which take place in a special monadic context MC, are the following:

$$C[\ e\ ] \parallel < \mathit{in}\ \mathit{out} > \rightarrow C[\ e'\ ] \parallel < \mathit{in}\ \mathit{out} > \ \mid e \mapsto e'\ ,$$

$$MC[\ \texttt{getString}\ ] \parallel < \mathit{string}\ \mathit{out} > \rightarrow MC[\ (\texttt{return}\ < \texttt{success}\ \mathit{string} >)\ ] \parallel < \Box\ \mathit{out} >\ ,$$

$$MC[\ \texttt{getString}\ ] \parallel < \texttt{eos}\ \mathit{out} > \rightarrow MC[\ (\texttt{return}\ < \texttt{failure}\ \texttt{"stream\_end"} >)\ ] \parallel < \Box\ \mathit{out} >\ ,$$

$$MC[\ (\texttt{putString}\ \mathit{string})\ ] \parallel < \mathit{in}\ \Box > \rightarrow MC[\ (\texttt{return}\ < \texttt{success}\ \texttt{"putString"} >)\ ] \parallel < \mathit{in}\ \mathit{string} >\ ,$$

$$MC[\ (\texttt{bind}\ (\texttt{return}\ \mathit{value})\ \mathit{cont\_fun})\ ] \parallel < \mathit{in}\ \mathit{out} > \rightarrow MC[\ (\mathit{cont\_fun}\ \mathit{value})\ ] \parallel < \mathit{in}\ \mathit{out} >\ ,$$

$$(\texttt{return}\ e) \parallel < \mathit{in}\ \mathit{out} > \rightarrow \texttt{done} \parallel < \mathit{in}\ \texttt{eos} >\ .$$

If we use a more convenient infix notation for the `bind` construct that uses $\gg\!=$ if the return value of the interaction is to be passed on to the continuation function, and $\gg$ otherwise, the monadic-style version of our login program takes the form

```
main = (putString "your name ?") >>
       getString >>=
       lambda u in
       (case
         < success name > -> ( putString < "hello" name >)
         otherwise ...
               u) >> ...
```

Other than for syntactic sugar, this program looks very much the same as the result continuation program since both concepts coincide if the interactions involved are just primitive `getString`s and `putString`s.

The advantage of this input/output programming model, as compared with streams and environment passing, is the higher level of abstraction. It is more declarative in the sense that the programmer does not have to worry about the details of stream handling, evaluation orders (strict versus non-strict), or uniqueness properties. All that remains to be done is to specify which interactions need to be performed and in which order. In this regard, monadic style programming closely resembles imperative programming, with decidedly more expressiveness and flexibility though. This is not very surprising since the bottom line is that it describes interactions with a state.

A.3 Interactions with a File System

As a concretization of monadic-style interactions, we will now have a brief look at the specification of input/output operations for an ordinary file system, as is known from UNIX, for instance. A file system consists of a set of files of which each either is empty or contains a sequence of elements (or entries) that can be located by their positions, usually counted by the number of entries relative to the beginning of the file. Entries selected by their positions can be read or overwritten. Prior to performing these elementary operations files must be opened, and afterwards they must be closed again. The files must be identified by names.

Interactions with a file system may be specified, as usual, by a set of transformation rules that are of the general form

$$MC[\, interaction \,] \parallel environment \mid guard \rightarrow \\ MC[\, (\texttt{return}\ value) \,] \parallel new\_environment \ .$$

The environment may be described by a tuple

$$environment = (N \rightarrow F,\ H \rightarrow I * N) \ ,$$

where N, F, H and I denote a set of file names, a set of file entries (or file contents), a set of handles on open files, and a set $\{0,\ 1,\ 2,\ \ldots\}$ of indices that enumerate file entries, respectively. The mapping $N \rightarrow F$ relates file names to file contents, and the mapping $H \rightarrow I * N$ defines handles as tuples of the form $(ind,\ file\_name)$ which identify specific entries in specific files. The contents of a file may simply be thought of as a list $< s_0\ \ldots\ s_i\ \ldots\ s_{n-1} >$ of n entries s_i, with i denoting the index positions.

The interactions that we wish to perform on this environment are `fopen` and `fclose` to open and close a file, and `fput`, `fget` and `fseek` to overwrite,

read and position the handle on a particular file entry, respectively.

$$MC[\,(\,\texttt{fopen}\ \mathit{name}\,)\,]\,\|\,(\,N \to F,\ H \to I{*}N\,)\mid \mathit{name} \notin N \wedge \mathit{handle} \notin H\ \to \\ MC[\,(\texttt{return}\ \mathit{handle})\,]\,\| \\ (N \to F \cup \{\mathit{name} \to <>\},\ H \to I * N \cup \{\mathit{handle} \to (0,\ \mathit{name})\})\ ,$$

$$MC[\,(\texttt{fopen}\ \mathit{name})\,]\,\|(N \to F,\ H \to I * N) \mid \mathit{name} \in N \wedge \mathit{handle} \notin H \to \\ MC[\,(\texttt{return}\ \mathit{handle})\,]\,\| \\ (N \to F,\ H \to I * N \cup \{\mathit{handle} \to (0,\ \mathit{name})\})\ ,$$

$$MC[\,(\texttt{fclose}\ \mathit{handle})\,]\,\|(N \to F,\ H \to I * N) \mid \\ \mathit{name} \in N \wedge \mathit{handle} \in H \wedge \mathit{handle} = (i,\ \mathit{name})\ \to \\ MC[\,(\texttt{return}\ \texttt{"closed"})\,]\,\| \\ (N \to F,\ H \to I * N \setminus \{\mathit{handle} \to (i,\ \mathit{name})\})\ .$$

There are two rules for `fopen`. The first one takes care of the case where the file does not yet exist and must be created, meaning that a new $\mathit{name} \to <>$ entry must be added to the mapping $N \to F$ and a new handle must be added to H. The second rule applies to the case where the file already exists, which requires a new handle only. In either case, the handle is initialized with position index 0 and returned as the value of the interaction. The interaction `fclose` takes as its argument the handle of an open file, removes it from H and returns the value `"closed"`.

The interactions with an open file are defined as follows:

$$MC[\,(\texttt{fget}\ \mathit{handle})\,]\,\|(N \to F,\ H \to I * N) \mid \\ \mathit{name} \in N \wedge \mathit{name} \to < \ldots s_i \ldots > \\ \wedge\, \mathit{handle} \in H \wedge \mathit{handle} = (i,\ \mathit{name}) \to \\ MC[\,(\texttt{return}\ s_i)\,]\,\|\,(N \to F,\ H' \to I' * N)$$

(in the new environment the handle has changed to $\mathit{handle}' \to (i{+}1,\ \mathit{name}) \in H' \to I' * N$);

$$MC[\,(\texttt{fput}\ \mathit{handle}\ ss_i)\,]\,\|(N \to F,\ H \to I * N) \mid \\ \mathit{name} \in N \wedge \mathit{name} \to < \ldots s_i \ldots > \\ \wedge\, \mathit{handle} \in H \wedge \mathit{handle} = (i,\ \mathit{name}) \to \\ MC[\,(\texttt{return}\ \texttt{"succ"})\,]\,\|\,(N \to F',\ H' \to I' * N)$$

(in the new environment the handle has again changed to $\mathit{handle}' \to (i + 1,\ \mathit{name}) \in H' \to I' * N$ and the file has changed in position i to $< \ldots\ ss_i\ \ldots > \in F'$);

$$MC[\,(\texttt{fseek}\ \mathit{handle}\ \mathit{index})\,]\,\|\mathit{handle} \in H \wedge \mathit{handle} = (i,\ \mathit{name}) \to \\ MC[\,(\texttt{return}\ \texttt{"succ"})\,]\,\|\,(N \to F,\ H' \to I' * N)$$

(in the new environment the handle has changed to $handle' \rightarrow (index,\ name) \in H' \rightarrow I' * N$).

And, of course, we also need a rule for each of the bind constructors $\gg=$ and $\gg$ that specify what happens to the return value of an interaction:

$$MC[\,(\texttt{return}\ value)\,]\ \gg= ff \parallel env \rightarrow MC[\,(ff\ value)\,] \parallel env\ ,$$

$$MC[\,(\texttt{return}\ value)\,]\ \gg ff \parallel env \rightarrow MC[\,ff\,] \parallel env\ .$$

As an example that illustrates the use of these interactions, consider a small program that opens a file named `"misc"`, positions the file handle at index position 668, reads the entry at this index, writes a string "aabbcc" into the next position and then closes the file again:

$$\begin{aligned} main = {} & (\texttt{fopen}\ \text{"}\texttt{misc}\text{"}) \gg= \texttt{lambda}\ hdl\ \texttt{in}\ (\texttt{fget}\ hdl\ 668) \gg \\ & (\texttt{fget}\ hdl) \gg= \texttt{lambda}\ entry\ \texttt{in}\ (\texttt{fput}\ hdl\ \text{"}\texttt{aabbcc}\text{"}) \gg \\ & (\texttt{fclose}\ hdl) \gg= \texttt{lambda}\ u\ \texttt{in}\ \texttt{if}(\texttt{eq}\ u\ \text{"}\texttt{closed}\text{"}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \texttt{then}\ entry\ \texttt{else}\ \ldots \end{aligned}$$

In this notation, the program looks almost like a sequence of UNIX system function calls that does the same job, the only difference being that some of the interactions have to be embedded in abstractions that explicitly pass along parameters such as the file handle or the entry read from the file, which in UNIX would be accomplished by means of side-effecting assignments to box variables. This is not very surprising insofar as the UNIX functions in fact realize almost the same abstractions that hide the low-level details of their implementation. They just need to be fed with the correct parameters and properly sequenced, everything else happens underground.

References

The contents of this appendix have in large parts been adopted from Claus Reinke's excellent PhD thesis [Rein98]. Streams were first proposed by Landin [Lan65], Henderson describes networks of functional processes communicating via streams [Hen80], followed later by several proposals for stream-based input/output for purely functional languages (see, for instance, Sect. 4 of [Sch93]). Hudak and Sundaresh introduced request/response streams as the input/output for HASKELL [HuSu89]. Environment passing was proposed by Achten and Plasmeijer as input/output mechanism for the functional language CLEAN [AcPl95]. Continuations were first introduced for semantic purposes, before Perry implemented them as the input/output mechanism of HOPE^C [Per91]. Finally, the concept of monad-based input/output for nonstrict functional languages is due to Peyton Jones and Wadler [PJWa93].

B

On Theorem Proving

The subject of this appendix is somewhat outside the scope of this textbook but it may help to underline the importance of machinery that lends formally sound support to interactively controlled symbolic computations involving free floating variables.

With growing size and complexity of programs, formal reasoning about the properties of at least their critical parts has become an issue of foremost concern. Expression-oriented languages appear to be particularly well suited for this purpose.

Reasoning about a piece of a program is a process that sets out with a theorem, also referred to as a goal in this context, about some property that is to be verified against some given theory. To do so, the process goes through sequences of proof steps that produce sequences of proof states. As each such state generally consists of a set of new goals that need to be proven, the entire proof process unfolds a tree structure of states. A single proof step is realized by what is called a tactic, which is a function that maps a single goal into a set of goals. If successful, this proof process terminates with states in which we have equality of two terms in each of the branches of the proof tree.

Some of the more important tactics include proof by contradiction, by induction, by rewriting according to an axiom or another theorem of the underlying theory, by reflexive equality of two terms, by unification of terms with respect to free variables and by (full) normalization.

The problem with this proof process is that, for practical reasons, it cannot be fully mechanized. For each proof state, it must be decided which of possibly several applicable tactics is the best one to proceed with. Even though the proof system could be made to offer suggestions as to some of the more promising tactics, the decision about which one to apply next to a particular (sub)goal ought to rest with the user in order to avoid explosion of the proof space. Also, it might become necessary to prove supplementary theorems (or lemmas) that need to be introduced to aid the original proof, which must be temporarily sidelined for this purpose. All this calls for a highly interactive proof system that at the user interface displays intermediate proof states in

high-level notation and gives the user an opportunity to select the (sub)goal (and possibly a subterm therein) and the tactic with which to perform the next step.

To illustrate how partly mechanized theorem proving should work, we consider, as a simple example, a function *rev* that is supposed to reverse the elements of a list. Using pattern matching as described in Chap. 11, this function may be specified in AL notation as[1]

$$
\begin{array}{l}
rev\ xs = (\texttt{case} \\
\qquad\qquad <>\ \ \| \texttt{true} \rightarrow <> \\
\qquad\qquad < u\ as[\ us\ ] >\ \ \| \texttt{true} \rightarrow ((append\ (rev\ us))\ u) \\
\qquad\quad \texttt{otherwise}\ xs \\
\qquad\quad\ xs)\ \ .
\end{array}
$$

For this function, we wish to prove the theorem

$$(rev\ (\ rev\ xs\ )\ )\ =\ xs\ \ ,$$

which says that applying *rev* twice to any list symbolized by the variable *xs* must return the list itself.

Since *rev* is a recursive function, proof by induction appears to be the natural choice. This proof breaks down into proving as subgoals the base case of an empty list

$$(rev\ (rev\ <>)) = <>\ \ ,$$

and the induction step

$$
\begin{array}{l}
(rev\ (rev\ list)) = list \Longrightarrow \\
\qquad\qquad (rev\ (rev\ ((append\ < u >)\ list))) = ((append\ < u >)\ list) \quad (*)
\end{array}
$$

for nonempty lists. This logical implication says that if the theorem holds for some list *list* it must also hold for the same list prepended by a single element *u*. It is important to note here that the variables *list* and *u* have been brought in from nowhere and may therefore be considered free in this implication, although in a strictly formal sense they are all-quantified since the implication must hold for all legitimate values substituted for them.

The first case can be proven by normalizing the left-hand side of the equation. It yields <>=<> and, after having established that both sides of the equation are truly equal, completes this part of the proof by returning the value `done`, say.

[1] Note that in this example we use the variable *append* simply as a name for a function that is assumed to append two argument lists without giving its implementation, and that we represent applications of *append* in curried form to facilitate transformation to customized constructor syntax as needed for the application of proof rules.

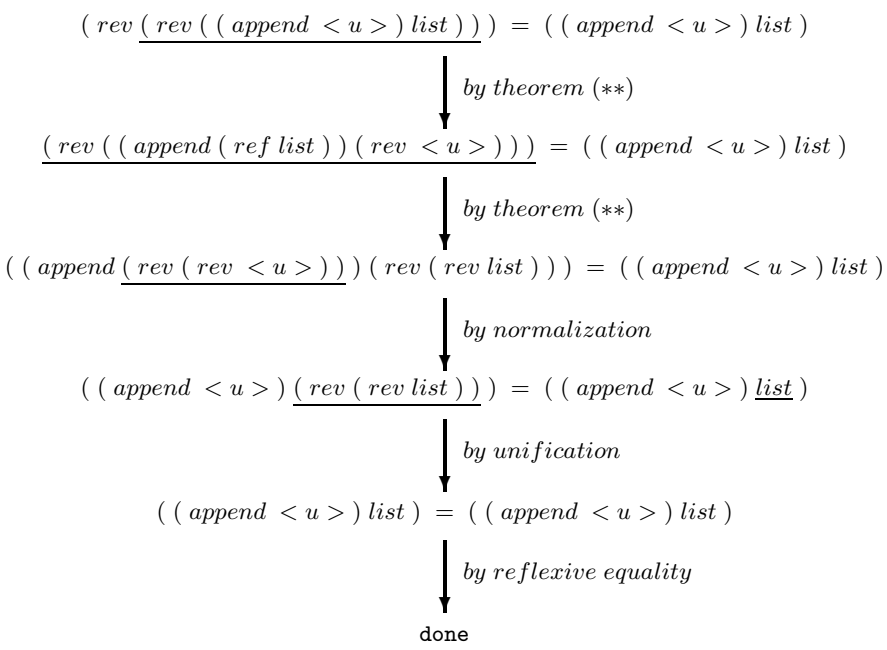


Fig. B.1. Sequence of proof steps for the theorem $(rev (rev\ xs)) = xs$

Proving the second case is a little more difficult. What needs to be shown is that the term on the right-hand side of the $\Longrightarrow$ sign is implied by the term on the left-hand side. To do so, we make use of another theorem

$$(rev\ ((append\ xs)\ ys)) = ((append\ (rev\ ys))(rev\ xs)) \quad , \quad (**)$$

which we assume is already proven. This theorem simply says that reversing a list made by appending a second sublist to a first sublist must result in the reversed first sublist appended to the reversed second sublist, which is rather straightforward.

The sequence of proofs and proof states for the right-hand side (or the conclusion) of the implication $(*)$ is depicted in Fig. B.1. It sets out with applications of this new theorem whose left-hand side matches the left-hand term of the conclusion twice in succession (the subterms that are affected by the proof steps are underlined). Substituting the matching (sub)terms by the properly instantiated right-hand sides of this theorem yields the proof state shown in the third line from the top. In the new left-hand term we can normalize the application $(rev\ (rev\ < u >))$ to $< u >$ to obtain as the proof state the equation in the fourth line from the top. Since the terms on both sides of the equation now feature matching structures, all that remains to be done is to unify both terms which, with respect to the free variables u and

list, yields $u = u$ and $(rev\ (rev\ list)) = list$, which is what was to be proved: the right-hand side of our induction $(*)$ is implied by its left-hand side.

Looking at the initial induction step and at the subsequent proof steps in Fig. B.1, we immediately recognize that only normalization can be done by λ-calculus machinery. All other steps involve major term transformations effected by tactics that obviously require another level of term representation on which the rewrite rules can be readily implemented. This level can be made available by pattern matching on customized constructor terms as described in Chap. 11. Working in the same system alternately with AL and constructor term representations calls for system functions that convert these representations into each other. Converting AL expressions into the meta-language of constructor terms in fact renders the terms constant with respect to AL interpretation. Since this is essentially a quoting mechanism, we may call the conversion function `quote` and its inverse `unquote`, as in LISP.

To define the `quote` function just for the terms that are involved in the above proof it suffices to consider the following fairly simple AL-like syntax:

$$term =_s var \mid \ < term\_1 \ldots term\_n > \ \mid \texttt{lambda}\ var\ \texttt{in}\ term \mid (term\_0\ term\_1) \mid term\_l = term\_r \ .$$

It includes variables, lists, abstractions, applications and equality of terms.

The `quote` function transforms these syntactical forms as follows:[2]:

$$\texttt{quote}[\![\ var\ ]\!] \rightarrow var[\ `var'\ ]$$

$$\texttt{quote}[\![\ < term\_1 \ldots term\_n >\ ]\!] \rightarrow < \texttt{quote}[\![\ term\_1\ ]\!] \ldots \texttt{quote}[\![\ term\_n\ ]\!] >$$

$$\texttt{quote}[\![\ \texttt{lambda}\ var\ \texttt{in}\ term\ ]\!] \rightarrow lam[\ `var'\ \texttt{quote}[\![\ term\ ]\!]\ ]$$

$$\texttt{quote}[\![\ (\ term\_0\ term\_1\ )\ ]\!] \rightarrow ap[\ \texttt{quote}[\![\ term\_0\ ]\!]\ \texttt{quote}[\![\ term\_1\ ]\!]\ ]$$

$$\texttt{quote}[\![\ (\ term\_l\ =_s\ term\_r\ )\ ]\!] \rightarrow equ[\ \texttt{quote}[\![\ term\_l\ ]\!]\ \texttt{quote}[\![\ term\_r\ ]\!]\ ] \ .$$

Quoting the theorem $(rev\ (rev\ xs)) = xs$ by this function thus yields the constructor term

$$equ[\ ap[\ var[\ `rev'\ ]\ ap[\ var[\ `rev'\ ]\ var[\ `xs'\ ]\ ]\ ]\ var[\ `xs'\ ]\ ] \ ,$$

which can now be subjected to transformations by pattern matching.

As an example, consider a function that realizes the induction tactic for theorems that relate to operations on lists. It must be applicable to the constructor term that represents the theorem and to the variable over which

[2] The inverse function `unquote` is defined in essentially the same way. It is recursively driven down into the subterms of a constructor term submitted as an argument to undo all the `quote`s.

induction is to be performed. It must return a list of two versions of the constructor term as its result. One version has the induction variable substituted by the empty list <> (the base case), and the other version has it substituted by the meta-language version

$$ap[\ ap[\ var[\ `append'\ ]\ <var[\ `u'\ ]>\ ]\ var[\ `list'\ ]\ ]$$

of $((append\ <u>)\ list)$ (the induction case). The latter term represents the right-hand side (or the conclusion) of the implication, with $`u'$ and $`list'$ being the names of free variables that enter the game out of thin air.

To keep things simple, we define the function *ind_tac* just for the induction case:

```
ind_tac = lambda term u in
          (case
            var[ v ] || ( eq u v ) →
                        ap[ ap[ var[ `append' ] <var[ `u' ] > ] var[ `list' ] ]
            var[ v ] || true → var[ v ]
             < term_1 ... term_n > || true →
                          < ( ind_tac term_1 u ) ... ( ind_tac term_n u ) >
            lam[ `v' term ] || true → lam[ `v' ( ind_tac term u ) ]
            ap[ term_0 term_1 ] || true →
                             ap[ ( ind_tac term_0 u ) ( ind_tac term_1 u ) ]
            equ[ term_l term_r ] || true →
                             equ[ ( ind_tac term_l u ) ( ind_tac term_r u ) ]
           otherwise term
              term ) .
```

The interesting pattern abstraction of this case construct that makes up the body of this function is the first one: whenever the constructor term submitted as an argument is a variable that equals the induction variable u, it is replaced by the construct $ap[\ ap[\ var[\ `append'\ ]\ <var[\ `u'\ ]>\ ]\ var[\ `list'\ ]\ ]$. All other pattern abstractions merely reconstruct the argument term as it is by doing nothing other than recursively driving *ind_tac* down into its subterms.

Thus, when applied as

$$(ind\_tac\ constr\_term\ `xs')\ ,$$

where *constr_term* fills in for the meta-language representation of our theorem, the function returns as its result

```
equ[ ap[ var[ `rev' ] ap[ var[ `rev' ]
          ap[ var[ `append' ] <var[ `u' ] > ] var[ `list' ] ] ] ]
                ap[ ap[ var[ `append' ] <var[ `u' ] > ] var[ `list' ] ] ] .
```

It is not very hard to see that unquoting this constructor term yields, as

expected, the right-hand side of the induction in AL notation (compare the first proof state in Fig. B.1):

$$(rev\ (rev\ ((append\ <u>)\ list))) = ((append\ <u>)\ list)\ .$$

Functions that realize rewrite tactics are more straightforward to generate. They may be composed of `case` constructs whose pattern abstractions implement the rewrite rules as they may be obtained more or less directly by quoting axioms and proven theorems of the underlying theory.

For instance, the single theorem (**) used in the first and second steps of the proof sequence in Fig. B.1 translates into

```
theo = lambda term in
       (case
          ap[ var[ 'rev' ] ap[ ap[ var[ 'append' ] xx ] yy ] ] || true →
          ap[ ap[ var[ 'append' ] ap[ var[ 'ref' ] yy ] ] ap[ var[ 'ref' ] xx ] ]
        otherwise term
                  term)
```

The other two steps of this proof sequence that need to be performed at the meta-language level are unification and the proof of equality of two terms.

Unification requires analyzing the structure of two terms and equating free variables occurring in one of the terms with (sub)structures in the other term. There is a well-established resolution/unification algorithm to this effect, proposed by Robinson, that is perfectly suited to implementation by pattern matching.

Equality of two terms, which in this particular case follows trivially from unification, may generally be established by literal comparison modulo α-conversion of variable occurrences, which too can best be done by pattern matching.

A complete semiautomated proof in a system that renders proof states visible to the user in AL-like notation thus goes repeatedly through the motions of

- inspecting in AL notation the entire proof state, i.e., the list of (sub)goals that need to be proven;
- selecting from this list a (sub)goal, a proof tactic and possibly the (sub)term of the goal to which the tactic is to be applied;
- applying the system function `quote` to the selected (sub)term if the proof tactic is something other than normalization;
- applying the chosen tactic function to the appropriate term representation;
- applying, if necessary, the system function `unquote` to the resulting proof state to reconstruct its AL representation.

This is done until either all subgoals generated in the course of the proof process have been dealt with or the proof fails somewhere, with none of the available tactics applicable.

The π–REDsystem of Chap. 10, in conjunction with the pattern-matching machinery of Chap. 11, is particularly well equipped to support such a proof system. It realizes the reduction semantics of a fully normalizing applied λ-calculus that in an orderly way deals with free variables, supports an interactively controlled stepwise execution mode and provides the means to decompile intermediate states of code execution into high-level AL-output. Such intermediate expressions may be freely modified, e.g., new applications and functions may be added, existing functions may be changed, old expressions, specifically applications, may be replaced by new ones, variables may be renamed or introduced out of the blue sky, their binding scopes may be redefined, etc., so that the meaning of the original expression may completely change. Moreover, reductions may be performed in any chosen order in subexpressions; referential transparency guarantees that this does not corrupt the determinacy of results.

The only things that are missing so far in this machinery are the conversion functions `quote` and `unquote` that transform AL expressions into the meta-language of customized constructor expressions and, after having them reduced by pattern matching, retransform them back into AL expressions. However, since both transformations are rather straightforward, merely requiring parsing the expressions to which they are applied, their implementation as built-in system functions is not at all difficult.

References

There is a large body of research on theorem proving. Some well-known theorem provers are those by Boyer and Moore [BoMo90] and the Larch Prover described by Garland and Guttag [GaGu91], with a wide variety of applications in circuit design and hardware/software (compiler) verification. A self-contained introduction to theorem proving with the Isabelle/HOL proof assistant is given by Nipkow, Paulson and Wenzel [NPW03], which in its first part covers elementary techniques of modeling and proving simple functional programs. A theorem prover for the functional language CLEAN which, interestingly enough, is written in CLEAN itself, is described in [dMvEPl02]. Robinson's unification algorithm is given in [Rob65]. The basics of mechanical theorem proving are covered in the textbook by Chang and Lee [ChLe73]. A textbook by Baader and Nipkov [BaNi99] gives a self-contained introduction to term rewriting, which is fundamental to theorem proving.

References

[ACCL90] Abadi, M.; Cardelli, L.; Curien, P.-L.; Levi, J.-J.: *Explicit Substitutions*, Proceedings of the 17th ACM Symposium on Principles of Programming Languages, ACM Press, 1990, pp. 1–16

[AS85] Abelson, H.; Sussmann, G.J.: *Structure and Interpretation of Computer Programs*, MIT Press and McGraw-Hill, 1985

[AcPl95] Achten, P.; Plasmeijer, R.: *The Ins and Outs of* CLEAN, Journal of Functional Programming, Vol. 5, No. 1, 1995, pp. 81–110

[AAM95] Aditya, S.; Arvind; Maessen, J.-W.; Augustsson, L.; Nikhil, R.S.: *Semantics of pH: a Parallel Dialect of Haskell*, MIT Laboratory for Computer Science, Computation Structure Group Memo 377-1, 1995

[ASU85] Aho, A.V.; Sethi, R.; Ullman, J.D.: *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1985

[AK91] Ait-Kaci, H.: *Warren's Abstract Machine: a Tutorial Reconstruction*, MIT Press, 1991

[Ama88] Amamiya, M.: *Data Flow Computing and Parallel Reduction Machine*, Future Generation Computer Systems, Vol. 4, No. 1, North Holland, 1988, pp. 53–67

[Ama91] Amamiya, M.: *An Ultra-multiprocessing Architecture for Functional Languages*, in: Gaudiot, J.-L.; Bic, L. (Eds.): Advanced Topics in Data-Flow Computing, Prentice Hall, 1991

[Amm81] Ammann, U.: *Code Generation of a Pascal Compiler*, in: Barron, D.W. (Ed.), Pascal – The Language and Its Implementation, Wiley, 1981

[Ant98] Antonakos, J.L.: *An Introduction to the Intel Family of Microprocessors*, Prentice Hall, 1998

[Ant99] Antonakos, J.L.: *The 68000 Microprocessor – Hardware and Software Principles & Applications*, 4th edition, Prentice Hall, 1999

[App99] Appel, W.; Ginsburg, M.: *Modern Compiler Implementation in C*, Cambridge University Press, 1999

[AGN96] Asperti, A.; Giovanetti, C.; Naletto, A.; *The Bologna Optimal Higher-Order Machine*, Journal of Functional Programming, Vol. 6, No. 6, 1996, pp. 763–810

[BaNi99] Baader, F.; Nipkow, T.: *Term Rewriting and All That*, Cambridge University Press, 1999

[Back72] Backus, J.: *Reduction Languages and Variable-Free Programming*, IBM Research Report RJ 1010, 1973

[Back73] Backus, J.: *Programming Language Semantics and Closed Applicative Languages*, Proceedings of the ACM Symposium on Principles of Programming Languages, Boston, Mass., 1973, pp. 71–86

[Bar84] Barendregt, H.P.: *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, Studies in Logic and the Foundations of Mathematics, 1984

[Ber75] Berkling, K.J.: *Reduction Languages for Reduction Machines*, Proceedings of the 2nd Annual Symposium on Computer Architecture, 1975, ACM/IEEE 75CH0916-7C, pp. 133–140

[Ber76] Berkling, K.J.: *A Symmetric Complement to the Lambda Calculus*, Internal Report GMD ISF-76-7, St. Augustin, Germany, 1976

[BeFe82] Berkling, K.J.; Fehr, E.: *A Consistent Extension of the Lambda-Calculus as a Base for Functional Programming Languages*, Information and Control 55, 1982, pp. 89–101

[Ber86] Berkling, K.J.: *Head-Order Reduction: a Graph Reduction Scheme for the Operational Lambda Calculus*, Lecture Notes in Computer Science, No. 254, Springer, 1986, pp. 26–48

[Ber94] Berkling, K.J.: *A "Fast" Representation of the Lambda Calculus*, Proceedings of the Workshop on Functional Programming JSSST'94, pp. 127–142

[Ber96] Berkling, K.J.: *The von Neumann-PLUS Architecture: an Evolutionary Development*, unpublished draft, 1996

[Ber97] Berkling, K.J.: Privately communicated handwritten notes, Syracuse, NY, Spring 1997

[BiWa88] Bird, R.; Wadler, P.: *Introduction to Functional Programming*, Prentice Hall, 1988

[Bird98] Bird, R.S.: *Introduction to Functional Programming with Haskell*, 2nd edition, Prentice Hall, 1998

[BoMo90] Boyer, S.R.; Moore, J.S.: *A Theorem Prover for a Computational Logic*, Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Computer Science, No. 449, Springer, 1990, pp. 1–15

[BrHa03] Bryant, R.E.; O'Hallaron, D.: *Computer Systems – a Programmer's Perspective*, Prentice Hall, 2003

[Bru72] deBruijn, N.G.: *A Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church–Rosser Theorem*, Indagationes Mathematicae 34, 1972, pp. 381–392

[Ca92] Cann, D.C.: *Retire Fortran? A Debate Rekindled*, Communications of the ACM, Vol. 35, No. 8, 1992, pp. 81–89

[Ca89] Cann, D.C.: *Compilation Techniques for High Performance Applicative Computation*, Technical Report CS-89-108, Lawrence Livermore Laboratory, Livermore, Ca., 1989

[CMQ83] Cardelli, L.; McQueen, D.: *The Functional Abstract Machine*, The ML/LCF/HOPE Newsletter, AT&T Bell Labs, Murray Hill, NJ, 1983

[Car84] Cardelli, L.: *Compiling a Functional Language*, Proceedings of the ACM Conference on LISP and Functional Programming, 1984, pp. 208–217

[Cam84] Campbell, J.A.: *Implementations of* PROLOG, Wiley, 1984

[ChLe73] Chang, C.-L.; Lee, R.C.-T.: *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, 1973

[Chu41] Church, A.: *The Calculi of Lambda Conversion*, Princeton University Press, 1941

[CCM85/87] Cousineau, G.; Curien, P.L.; Mauny, M.: *The Categorial Abstract Machine*, Proceedings of the Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science, No. 201, Springer, 1985, pp. 50–64, and Science of Computer Programming, No. 8, 1987, pp. 173–202

[ClKo82] Clark, R.; Koehler, S.: *The UCSD Handbook*, Prentice Hall, 1982

[Cre90] Crégut, P.: *An Abstract Machine for the Normalization of λ-Terms*, Proceedings of the ACM Conference on LISP and Functional Programming, 1990, pp. 333–340

[Cur29] Curry, H.B.: *An Analysis of Logical Substitution*, American Journal of Mathematics, No. 51, 1929, pp. 363–384

[Cur36] Curry, H.B.: *First Properties of Functionality in Combinatory Logic*, Tohoku Mathematical Journal, No. 41, 1936, pp. 371–401

[DHS00] Diehl, S.; Hartel, P.; Sestoft, P.: *Abstract Machines for Programming Language Implementation*, Future Generation Computer Systems, Vol. 16, No. 7, 2000, pp. 739–751

[Dyb87] Dybvik, R.K.: *The* SCHEME *Programming Language*, Prentice Hall, 1987

[FW87] Fairbairn, J.; Wray, S.: *TIM: a Simple Lazy Abstract Machine to Execute Supercombinators*, Proceedings of the Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science, No. 274, Springer, 1987, pp. 34–45

[FWH92] Friedman, D.P.; Wand, M.; Haynes, C.T.: *Essentials of Programming Languages*, MIT Press and McGraw-Hill, 1992

[FGSW03] Friedman, D.P.; Ghuloum, A.; Siek, J.G.; Winebarger, L.: *Improving the Lazy Krivine Machine*, Technical Report TR 581, Dept. of Computer Science, Indiana University, Bloomington, Indiana, 2003; also to appear in: Journal on Higher-Order and Symbolic Computation, Kluwer

[GK96] Gaertner, D.; Kluge, W.E.: *π-*RED$^+$ *– an Interactive Compiling Graph Reduction System for an Applied λ-Calculus*, Journal of Functional Programming, Vol. 6, No. 5, 1996, pp. 723–757

[GaGu91] Garland, S.J.; Guttag, J.V.: *A Guide to LP, the Larch Prover*, Technical Report, MIT Laboratory for Computer Science, and Technical Report No. 82, DEC System Research Center, 1991

[GoRo89] Goldberg, A.; Robson, D.: *Smalltalk 80 – The Language and its Implementation*, Addison-Wesley, 1989

[Gol94] Goldson, D.: *A Symbolic Calculator for Nonstrict Functional Programs*, The Computer Journal, Vol. 37, No. 3, 1994, pp. 177–187

[GrLe02] Grégoire, B.; Leroy, X.: *A Compiled Implementation of Strong Reduction*, Proceedings of the ACM International Conference on Functional Programming, 2002, pp. 235–246

[Gre01] Grelck, C.: *Implicit Shared Memory Multiprocessor Support for the Functional Language SAC*, PhD thesis, University of Kiel, Germany, 2001, published by Logos Verlag, ISBN 3-89722-719-3

[Gre03] Grelck, C.: *A Multithreaded Compiler Backend for High-Level Array Programming*, Proceedings of the International Conference on Parallel and Distributed Computing and Networks, ISBN 0-88986-341-5, ISSN 1027-2666, Innsbruck (Austria) 2003, pp. 478–484

[GuKiWa85] Gurd, J.R.; Kirkham, C.C.; Watson, I.: *The Manchester Prototype Dataflow Computer*, Communications of the ACM, Vol. 28, No. 1, 1985, pp. 34–52

[HMP98] Hardin, T.; Maranget, L.; Pagano, B.: *Functional Runtime Systems Within the λσ-Calculus*, Journal of Functional Programming, Vol. 8, No. 2, 1998, pp. 131–176

[Har92] Harel, D.: *Algorithmics – the Spirit of Computing*, 2nd edition, Addison-Wesley, 1992

[HaMi99] Hammond, K.; Michaelson, G. (Eds): *Research Directions in Parallel Functional Programming*, Springer, 1999

[HaRi99] Hansen, M.R.; Rischel, H.: *Introduction to Programming Using SML*, Addison-Wesley, 1999

[HaHe95] Harman, T.L.; Hein, D.T.: *The Motorola MC68000 Microprocessor Family: Assembler Language Interface Design & System Design*, Pearson Education, 1995

[Hen80] Henderson, P.: *Functional Programming: Application and Implementation*, Prentice Hall, 1980

[Hil90] Hilton, M.L.: *Implementation of Declarative Languages*, PhD thesis, CASE Center Technical Report No. 9008, Syracuse University, Syracuse, NY, 1990

[Hin69] Hindley, J.R.: *The Principal Type-Scheme of an Object in Combinatory Logic*, Transactions of the American Mathematical Society, Vol. 146, 1969, pp. 29–60

[HS86] Hindley, J.R.; Seldin, J.P.: *Introduction to Combinators and λ-Calculus*, Cambridge University Press, London Mathematical Society Student Texts, 1986

[HuSu89] Hudak, P.; Sundaresh, R.S.: *On the Expressiveness of Purely Functional I/O Systems*, Technical Report, Dept. of Computer Science, Yale University, New Haven, 1989

[Hudall92] Hudak, P.; Peyton Jones, S.L.; Wadler, P.L.; Arvind; Boutel, B.; Fairbairn, J.; Fasel, J.; Guzman, M.; Hammond K.; Hughes, J.; Johnsson, T.; Kieburtz, R.; Nikhil, R.S.; Partain, W.; Peterson, J.: *Report on the Functional Programming Language* HASKELL*, Version 1.2*, SIGPLAN Notices 27, 1992

[Hu82a] Hughes, R.J.M.: *Super-Combinators – a New Implementation Technique for Applicative Languages*, Proceedings of the ACM Conference on LISP and Functional Programming, Pittsburgh, Pa., 1982, pp. 1–10

[Joh84] Johnsson, T.: *Efficient Compilation of Lazy Evaluation*, ACM Conference on Compiler Construction, Montreal, Que., 1984, pp. 58–69

[Joh85] Johnsson, T.: *Lambda Lifting: Transforming Programs to Recursive Equations*, Proceedings of the Conference on Functional Programming and Computer Architecture, Lecture Notes in Computer Science, No. 201, Springer, 1985, pp. 190–203

[Joh87] Johnsson, T.: *Compiling Lazy Functional Languages*, PhD thesis, Chalmers University of Technology, Goeteborg, 1987

[Jo99] Jones, R.: *Garbage Collection – Algorithms for Automated Dynamic Memory Management*, Wiley, 1999

[Kap94] Kaplan, R.M.: *Constructing Language Processors for Little Languages*, Wiley, 1994

[Kle36] Kleene, S.C.: *General Recursive Functions of Natural Numbers*, Mathematical Annals, No. 112, 1936, pp. 727–742

[Kge79] Kluge, W.E.: *The Architecture of the Reduction Machine Hardware Model*, Internal Report GMD ISF-79-3, St. Augustin, Germany, 1979

[KS80] Kluge, W.E.; Schluetter, H.: *An Architecture for Direct Execution of Reduction Languages*, Proceedings of the International Workshop on High-Level Language Computer Architecture, Fort Lauderdale Fla., 1980, pp. 174–180

[Kge92] Kluge, W.E.: *The Organization of Reduction, Data Flow, and Control Flow Systems*, MIT Press, 1992

[Kge94] Kluge, W.E.: *A User's Guide for the Reduction System* π-RED, Report Nr. 9419, Dept. of Computer Science, University of Kiel, Germany, 1994

[Kog91] Kogge, P.M.: *The Architecture of Symbolic Computers*, McGraw-Hill, 1991

[Kri85] Krivine, J.-L.: *Un interpréte du λ-calcul.*, 1985

[Lan64] Landin, P.J.: *The Mechanical Evaluation of Expressions*, The Computer Journal, Vol. 6, No. 4, 1964, pp. 308–320

[Lan65] Landin, P.J.: *A Correspondence Between* ALGOL 60 *and Church's Lambda-Notation*, Communications of the ACM, Vol. 8, No. 2/3, 1965, pp. 89–101 and 158–165

[Ler90] Leroy, X.: *The Zinc Experiment: an Economical Implementation of the ML Language*, Technical Report No. 117, INRIA Rocquencourt, France, 1990

[Lev94] Levine, F.: *RISC System/*6000 *PowerPC System Architecture*, Morgan Kaufmann, 1994

[LiYe99] Lindholm, T.; Yellin, F.: *The* JAVA^{TM} *Virtual Machine Specification*, Addison-Wesley, 1999

[LiWa93] Livadas, P.E.; Ward, C.: *Computer Organization and the MC68000*, Prentice Hall, 1993

[Ma96] Mark, R.: *Writing Compilers and Interpreters*, Wiley, 1996

[McCA65] McCarthy, J.; Abrahams, P.W.; Edwards, D.J.; Hart, T.P.; Levin, M.I.: *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass., 1965

[Mil78] Milner, R.: *A Theory of Type Polymorphism in Programming*, Journal of Computer and System Sciences, Vol. 17, 1978, pp. 348–375

[MTHQ97] Milner, R.; Tofte, M.; Harper, R.; MacQueen, D.: *The Definition of Standard ML*, MIT Press, 1997

[dMvEPl02] de Mol, M.; van Eekelen, M.; Plasmeijer, R.: *Theorem Proving for Functional Programmers*, Proceedings of the International Workshop on the Implementation of Functional Languages, Lecture Notes in Computer Science, No. 2312, Springer, 2002, pp. 55–71

[NPW03] Nipkow, T.; Paulson, L.C.; Wenzel, M.: *Isabelle HOL – A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science, No. 2283, Springer, 2003

[PaCu90] Papadopoulos, G.M.; Culler, D.E.: MONSOON*: an Explicit Token-Store Architecture*, Proceedings of the 17th Annual Symposium on Computer Architecture, ACM, 1990, pp. 82–91

[PaHe90] Patterson, D.A.; Hennessy, J.L.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990

[PeDa82] Pemberton, S.; Daniels, M.: PASCAL *Implementation, the P4 Compiler*, Ellis Horwood, 1982

[Per91] Perry, N.: *The Implementation of Practical Functional Programming Languages*, PhD thesis, Imperial College, London, 1991

[PeyJ87] Peyton Jones, S.L.: *The Implementation of Functional Programming Languages*, Prentice Hall, 1987

[PeSa89] Peyton Jones, S.L.; Salkild, J.: *The Spineless Tagless G-Machine*, ACM Proceedings of the Conference on Functional Programming Languages and Computer Architecture, London, 1989, pp. 184–201

[PeyJ92] Peyton Jones, S.L.: *Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine*, Journal of Functional Programming, Vol. 2, No.2, 1992, pp. 127–202

[PJWa93] Peyton Jones, S.L.; Wadler, P.: *Imperative Functional Programming*, Proceedings of the 20th Symposium on Principles of Programming Languages, ACM, 1993

[PvE93] Plasmeijer, R.; van Eekelen, M.: *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley, 1993

[Pau99] Paul, R.: *SPARC Architecture Assembly Language*, 2nd edition, Prentice Hall, 1999

[Pau96] Paulson, L.C.: ML *for the Working Programmer*, Cambridge University Press, 1996

[Post43] Post, E.: *Formal Reductions of the General Combinatorial Decision Problem*, American Journal of Mathematics, No. 65, 1943, pp. 197–215

[RR64] Randell, B.; Russel, L.J.: *Algol 60 Implementation*, Academic Press, 1964

[RS92] Rathsack, C., Scholz, S.-B.: *Lisa – a Lazy Interpreter for a Full-Fledged λ-Calculus*, Proceedings of the International Workshop on the Implementation of Functional Languages, RWTH Aachen, Germany, 1992

[Rea89] Reade, C.: *Elements of Functional Programming*, Addison-Wesley, 1989

[Rein98] Reinke, C.: *Functions, Frames, and Interactions*, PhD thesis, Report Nr. 9804, Dept. of Computer Science, University of Kiel, Germany, 1998

[Rob65] Robinson, J.A.: *A Machine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM, No. 12, 1965, pp. 23–41

[RoSi82] Robinson, J.A.; Sibert, E.E.: LOGLISP*: an Alternative to* PROLOG, Machine Intelligence, No. 10, Ellis Horwood, 1982, pp. 399–419

[SBK92] Schmittgen, C.; Bloedorn, H.; Kluge, W.: π-RED* *– a Graph Reducer for a Full-Fledged λ-Calculus*, New Generation Computing, Vol. 10, No. 2, 1992, pp. 173–195

[Scho24] Schoenfinkel, M.: *Ueber die Bausteine der Mathematischen Logik*, Mathematische Annalen, Vol. 92, 1924, pp. 305–316

[Scho03] Scholz, S.-B.: *Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting*, Journal of Functional Programming, Vol. 13, No. 6, 2003, pp. 1005–1059

[Sch93] Schreiner, W.: *Parallel Functional Programming, an Annotated Bibliography*, 2nd edition, Technical Report, Research Institute for Symbolic Computation, University of Linz, Austria, 1993

[SSB01] Staerk, R.F.; Schmid, J.; Boerger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer, 2001

[Ste84] Steele, G.L.: *Common* LISP *– The Language*, Digital Press, 1984

[Tho96] Thompson, S.: HASKELL *– the Craft of Functional Programming*, Addison-Wesley, 1996

[Trou93] Troullinos, N.B.: *Head-Order Techniques and Other Pragmatics of Lambda Calculus Graph Reduction*, PhD thesis, CASE Center Technical Report No. 9322, Syracuse University, Syracuse, NY, 1993

[Tur37] Turing, A.: *On Computable Numbers with an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, No. 42, 1937, pp. 230–265

[Tur85] Turner, D.A.: *Miranda – a Non-strict Functional Language with Polymorphic Types*, Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, No. 201, Springer, 1985, pp. 1–16

[Tur79] Turner, D.A.: *A New Implementation Technique for Applicative Languages*, Software Practice and Experience, Vol. 9, No. 1, 1979, pp. 31–49

[Ull98] Ullman, J.U.: *Elements of* ML *Programming*, 2nd edition, Prentice Hall, 1998

[Wads71] Wadsworth, C.P.: *Semantics and Pragmatics of the Lambda Calculus*, PhD thesis, Oxford University, 1971

[Wea93] Weaver, D.L.: *SPARC Architecture Manual Version* 9, Prentice Hall, 1993

[WeSm94] Weiss, S.; Smith, J.E.: *POWER and PowerPC*, Morgan Kaufmann, 1994

[Wi96] Wirth, N.: *Compiler Construction*, Addison-Wesley, 1996

[ZB89] Zhang, S.; Berkling, K.: *The Soundness and Completeness of Head-Order Reduction*, CASE Center Technical Report No. 8907, Syracuse University, Syracuse, NY, 1989

[Sun90] *Sun Common* LISP *4.0 User's Guide*, Sun Microsystems, 1990

[All92] *Allegro CL User Guide*, Vol. 1, Version 4.1, Franz Inc., 1992

[PLTS96] *PLT DrScheme: Programming Environment Manual*, `http://download.plt-scheme.org/doc/drscheme`, 1996 (last update 2004)

Index