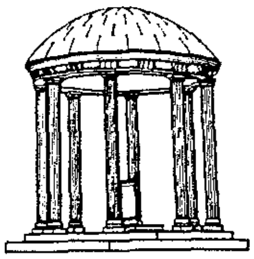
Graph Reduction Without Pointers

TR89-045

December, 1989

William Daniel Partain



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Graph Reduction Without Pointers

by

William Daniel Partain

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill, 1989

Approved by:

Gyula Magó

Gyula A. Magó, advisor

Jan F. Prins

Jan F. Prins, reader

[illegible]

Donald F. Stanat, reader

WILLIAM DANIEL PARTAIN. Graph Reduction Without Pointers (Under the direction of Gyula A. Magó.)

Abstract

Graph reduction is one way to overcome the exponential space blow-ups that simple normal-order evaluation of the lambda-calculus is likely to suffer. The lambda-calculus underlies lazy functional programming languages, which offer hope for improved programmer productivity based on stronger mathematical underpinnings. Because functional languages seem well-suited to highly-parallel machine implementations, graph reduction is often chosen as the basis for these machines' designs.

Inherent to graph reduction is a commonly-accessible store holding nodes referenced through "pointers," unique global identifiers; graph operations cannot guarantee that nodes directly connected in the graph will be in nearby store locations. This absence of locality is inimical to parallel computers, which prefer isolated pieces of hardware working on self-contained parts of a program.

In this dissertation, I develop an alternate reduction system using "suspensions" (delayed substitutions), with terms represented as trees and variables by their binding indices (de Bruijn numbers). Global pointers do not exist and all operations, except searching for redexes, are entirely local. The system is provably equivalent to graph reduction, step for step. I show that if this kind of interpreter is implemented on a highly-parallel machine with a locality-preserving, linear program representation and fast scan primitives (an FFP Machine is an appropriate architecture) then the interpreter's worst-case space complexity is the same as that of a graph reducer (that is, equivalent sharing), and its time complexity falls short on only one unimportant case. On the other side of the ledger, graph operations that involve chaining through many pointers are often replaced with a single associative-matching operation. What is more, this system has no difficulty with free variables in redexes and is good for reduction to full beta-normal form.

These results suggest that non-naive tree reduction is an approach to supporting functional programming that a parallel-computer architect should not overlook.

Preliminaries

Programming language. I support some of my descriptions by showing an implementation encoded in Standard ML, set in a sans serif font. Appendix A is a reader's introduction to ML and defines the supporting routines for the programs in the dissertation proper. I chose ML because the compiler from Bell Laboratories [8] was the best-implemented functional language available to me. The code shown is directly extracted from working programs.

A stylistic matter. I veer from the usual habit of calling myself "we," siding with E. B. White:

> It is almost impossible to write anything decent using the editorial "we," unless you are the Dionne family. Anonymity, plus the "we," gives a writer a cloak of dishonesty, and he finds himself going around, like a masked reveler at a ball, kissing all the pretty girls [84, page 121].

Acknowledgments. I am not sure how I got into the Ph.D. business, but I know how I got through it. The Team Magó members have been the best of colleagues, notably David Middleton, initiator into the Mysteries, Edoardo Biagioni, with his startling imagination, and Bruce Smith, with his breadth of understanding and good sense. Charles Molnar and his group added more than a little spice through their collaboration with us. Vern Chi and the Microelectronics Systems Lab have provided superb computing facilities, even if I was mainly an intruder. "Net people" provided many small helps and assurances; Paul Watson went beyond the call duty by sending a copy of his hard-to-get thesis. Bharat Jayaraman and Rick Snodgrass, former committee members, read drafts even after they had decided to leave; that is conscientiousness! György Révész provided a needed boost during his visit from the T. J. Watson Research Center. As for sanity, my family and friends have served admirably as bouncers for Dorothea Dix hospital, including my father who has hounded me mercilessly about finishing and my mother who made a point not to. Ed McKenzie was as good a lunch crony as one could hope to find, but I will never forgive him for completing his degree in only four years. Phil and Paige LeMasters failed to disguise their deliberate effort to keep me in contact with the world outside Sitterson Hall.

My noble and ennobling committee—Gyula Magó, David Plaisted, Jan Prins, Don Stanat and Jennifer Welch—have bravely weathered the drafts from room 327 and have vastly improved my material. Prof. Stanat lured me into the Department and has stood behind his mistakes, even as I flamed the faculty and wrote purple prose into proposals. Prof. Magó has enhanced his reputation as the best thesis advisor in the Department, unbuffeted by the fashions of graduate-student whims, tolerant of not-always-serious meetings, readily available for consultation, and incisive (but not opaque) in his critique. I offer my heartfelt thanks to each one.

I am grateful to the U. S. Army Research Office which provided financial support for this work through an Army Science and Technology Fellowship (grant number DAAL03-86-G-0050).

Comments. I welcome your comments and corrections. My e-mail address is `partain@cs.unc.edu`, and paper mail will reach me via the Computer Science Department, UNC, Sitterson Hall, Chapel Hill, NC 27599-3175. Electronic comments sent to Prof. Magó (`mago@cs.unc.edu`) will be forwarded to me even after I leave UNC.

Contents

1 The problem **1**

1.1 Motivation . . . 1

1.2 Thesis statement . . . 9

1.3 Dissertation organization . . . 10

2 The λ-calculus **12**

2.1 Syntax . . . 13

2.2 Computing with the λ-calculus . . . 15

2.3 Other λ-calculus reduction rules . . . 16

2.4 β-normal form and normal-order reduction . . . 17

2.5 Other normal forms and evaluation orders . . . 18

2.6 The name-free λ-calculus . . . 20

2.7 Combinators . . . 22

2.8 The practical use of the λ-calculus . . . 26

2.9 The necessity of sharing for normal-order evaluation . . . 27

3 Graph reduction: the λ_g-interpreter **30**

3.1 Graph structure and terminology . . . 32

3.2 Finding a redex and β_g-reduction . . . 35

3.3 Sharing free expressions and lazy copying . . . 38

3.4 More on variable bindings . . . 40

3.4.1 Graph reduction with binding indices . . . 40

3.4.2 Wadsworth's use of backpointers . . . 42

3.5 Graph-reduction architectures and sharing . . . 45

3.6 Graph rewriting . . . 51

4 Reduction with suspensions: the λ_s-interpreter **53**

4.1 λ_s-term structure and terminology . . . 58

4.2 β_s-reduction: the β_s rule . . . 63

4.3 Searching for the next redex 66
4.4 Lazy copying of shared rators 75
4.5 Tidying λ_s-terms 76
4.5.1 Removing useless suspensions 79
4.5.2 Removing trivial suspensions 80
4.5.3 Moving λ_s-abstractions above suspensions 81
4.5.4 Rotating suspensions 84
4.5.5 Upward and downward s-connections 85
4.5.6 Constraints on moving suspensions 86
4.5.7 Tidying: definition and important properties 86
4.5.8 The recurring example on the λ_s-interpreter 90
4.6 α-equivalence of λ_s-terms 90
4.7 Equivalence to graph reduction: correctness 92
4.8 Related approaches to λ-calculus evaluation 94
4.8.1 Efforts to find simpler reduction rules 94
4.8.2 Comparison with environment-based evaluation 97
4.8.3 Environment/reduction hybrids 100
4.8.4 Pointers versus λ_s-pointers 100

5 The λ_s-interpreter on an FFP Machine 102
5.1 Introduction to the FFP Machine 103
5.1.1 Project history and design goals 103
5.1.2 Basic structure of an FFP Machine 104
5.1.3 Communication and partitioning 106
5.1.4 Broadcasting and sorting operations 108
5.1.5 Single-result operations 109
5.1.6 Scan, or parallel prefix, operations 109
5.1.7 Computing level numbers 110
5.1.8 Calculating exotic level numbers 112
5.1.9 Calculating selectors and first/last bits 113
5.1.10 Low-level programming style in an FFP Machine . . . 114
5.1.11 Copying and sharing in an FFP Machine 115
5.1.12 Related non-graph-reduction architectures 116
5.2 An implementation of a λ_s-interpreter 119
5.2.1 Basic algorithms 120
5.2.2 Controlling the interpreter 125
5.2.3 Reducing to β-normal or root-lambda form 126
5.2.4 β_s-reduction and local tidying 130
5.2.5 Nonlocal tidying 132

5.2.6 Global checking 133
5.2.7 Summary of the FFP Machine implementation 138
5.3 Equivalence to graph reduction: efficiency 140
5.3.1 Space complexity 141
5.3.2 Time complexity 142
5.4 Previous FFP Machine implementations of the λ-calculus . . 151

6 Embellishments 153
6.1 Suspension lists 153
6.2 Dealing with recursion 156
6.3 Exotic suspensions 157
6.4 More parallelism 158
6.5 Ordering in λ_s-terms and supercombinators 158
6.6 Speculative copying 160
6.7 Going further with binding indices 161

7 Conclusions 163

A Programming with ML 166
A.1 The one-minute ML programmer 166
A.2 Utility functions 169
A.2.1 λ_s-term functions 169
A.2.2 λ_g-graph functions 174

Bibliography 178

Index 201

List of Tables

4.1 Comparison of β_g- and β_s-reduction 65
4.2 Comparison of λ_g- and λ_s-lazy copying 77
4.3 λ_s-interpreter rule summary 89
4.4 Staples's "graph-like lambda calculus" rules 95
4.5 Révész's reduction rules 97

5.1 Calculating the standard level numbers 112
5.2 Comparison of asymptotic time complexities 143

List of Figures

2.1 λ-abstractions 14
2.2 λ-applications 15
2.3 β-reductions 16
2.4 A λ-term in β-normal form 17
2.5 A λ-term in head-normal form 18
2.6 How to capture a variable in one easy lesson 20
2.7 Reduction with binding indices 21
2.8 The recurring example: five tree reduction steps 24
2.9 Function composition example 29

3.1 Simple graph reduction 31
3.2 Graph nodes' structure 33
3.3 Graph reduction with copying 38
3.4 Graph reduction with a shared MFE 39
3.5 A lazy copy 39
3.6 Lazying copying will not work for non-weak reduction 42
3.7 The λ_g-interpreter on the recurring example 43
3.8 Graph reduction with the name-free λ-calculus 44
3.9 Wadsworth's use of backpointers and indirection nodes ($\leadsto$) . 44
3.10 ALICE machine organization ("dancehall") 46
3.11 Flagship machine organization ("boudoir") 47

4.1 Graph reduction vs. reduction with suspensions 55
4.2 A λ_g-interpreter step vs. a λ_s-interpreter step 56
4.3 A λ_s-term with a suspension at its root 59
4.4 Definition of a Term 59
4.5 An example with much λ_s-term notation 60
4.6 β_s-reduction of a shared redex 64
4.7 A term after one β_g- or β_s-reduction 68
4.8 The three cases of pointer-following 69

4.9 Example distribution of λ_s-pointers in a λ_s-term 71
4.10 A lazy copy in the λ_s-interpreter 76
4.11 Pointer-following run amok 78
4.12 β-reduction followed by [triv-ptee] rule 81
4.13 λx rator far from the plain-node λ_s-application above 82
4.14 The [λ-up] rule and its graph-reduction equivalent 83
4.15 Suspension reordering needed 84
4.16 Rotate adjacent suspensions leftward 84
4.17 The [sus-rotl] rule preserves λ_{sg}-equivalence 85
4.18 Kinds of s-connections 86
4.19 How far can suspension $[x]$ be moved? 87
4.20 The λ_s-interpreter on the recurring example 91
4.21 Equivalent, non-identical λ_s-terms 92
4.22 Staples's $\sigma\alpha$-rules 96
4.23 An example of director strings 100

5.1 Structure of an FFP Machine 105
5.2 Partitioning in an FFP Machine 107
5.3 Nesting levels for a λ_s-term 111
5.4 λ_s-term with its selectors shown 113
5.5 Non-local tidying 132
5.6 The FFP Machine interpreter on the recurring example . . . 139
5.7 A λ_s-term stuffed with non-plain nodes 142
5.8 Unpleasant example of looking for redex 144
5.9 Last-instance relocation in straight-line code 146
5.10 An example with unbounded last-instance relocations 147
5.11 Impossible sequence of [sus-rotl]'s 149

6.1 Left-skewed suspensions become a suspension list 154
6.2 Binding indices change in a suspension list 155
6.3 Suspension lists collapse into their parent list 155
6.4 λ-application list and subsequent reduction 156
6.5 Two Y combinator reductions, graphs and suspensions 157
6.6 Mutually recursive functions in a suspension list 157
6.7 Transforming to supercombinators, with a suspension list . . 159
6.8 Abstracting a free expression 160
6.9 Binding indices on suspensions and λ-applications 162

Chapter 1

The problem

Von Neumann languages constantly keep our noses pressed in the dirt of address computation and the separate computation of single words, whereas we should be focusing on the form and content of the overall result we are trying to produce.

— John Backus, "The History of FORTRAN I, II, and III" (1978).

The broad concern behind this dissertation is how to implement functional programming languages on highly parallel computers without recourse to graph reduction that uses pointers into a global memory. This chapter sets out the specific problem that I examine and why it is important. I then present my thesis and sketch the plan of attack.

In this chapter, I presume some knowledge of the λ-calculus, graph reduction, and the FFP Machine; Chapters 2 and 3 and Section 5.1 introduce these topics, respectively. Less-important unfamiliar terms may be traced through the index.

1.1 Motivation

Two major emphases in computing are the quest for faster machines and the search for more productive, less error-prone ways to program them.

Computers today are some four orders of magnitude faster than the earliest machines built around the time of von Neumann's original proposal for the stored-program serial computer in 1946 [39]. Improved technologies account for much of the speedup, as electro-mechanical parts have given way to sub-micron VLSI chips. Faster parts built with better technologies will continue to appear—but not indefinitely. Meanwhile, the appetite for more speed will continue unsatisfied.

Why not improve computing speeds by using *many* processors at once to solve a problem? This idea dates back to the earliest computing days: for example, Univac lauded the "super-parallelism" of its Larc system, which could have two processors (1956) [63]; on the software side, the first article in the first issue of the British *Computer Journal* was about "Parallel Programming" (1958) [72]. Now, the use of many processors to achieve greater speeds is unavoidable, as the marginal cost of a faster uniprocessor ("serial MIPS") is high whereas the cost of a boxful of VLSI microprocessors (potential "parallel MIPS") continues to decrease. As a result, many multiprocessor machines have been built, and some are commercially available. Many of these designs have a modest number of processors and run separate programs on separate processors; typically, all processors share a global memory. These machines' selling point is cost-effectiveness. In contrast, my concern is with raw speed on an individual problem that has enough parallelism, and I limit myself to machines that deploy many processors to this end. (I exclude pipelined vector processors, because they offer only limited speedup.)

On the software side, thousands of people have worked to make programming a more productive human endeavor. Higher-level languages, structured programming, and strong type-checking are among the tools and techniques used. Yet we still have a "software crisis" revealed by error-ridden code, by programs tenuously related to their specifications, and by bloated software projects, years behind schedule. The field of software engineering is dedicated to surmounting the crisis.

Programming is even harder for multiprocessors. Most importantly, an additional kind of error, the "timing bug," enters the picture. The order in which pieces of a program (on different processors) synchronize with each other may vary from run to run. Instrumenting one's code to smoke out the bugs may change the timing enough that they vanish (the "probe effect"). Correct answers on one run provide no assurance that the program's timing is right; deadlocks may suddenly arise, perhaps when ramping up to larger-scale production work. In their report, "Exploiting Multiprocessors: Issues and Options," McGraw and Axelrod make clear the severity of timing problems in practice [150]. Alan Karp subtitles his review of tools for parallel programming as "The state of the art of parallel programming and what a sorry state that art is in" [112].

Tasks in a parallel program must communicate with each other, to share data and to synchronize their actions. How can a program and data be mapped onto processors so communication is efficient and parallel operations are not delayed? Moreover, how does one re-balance the load across the processors as the program's requirements change during execution?

Some of the approaches to the thorniness of parallel programming are instructive. Perhaps the most common way multiprocessors are programmed is with low-level, error-prone tools (e.g., extended FORTRAN)—but only for "well-behaved" problems with static data structures and predictable run-time execution profiles. Happily, many important scientific programs fit this mold. A person plans the mapping of program and data to processors, and the results can be good: the work that yielded the impressive Sandia Labs speedups is in this category [71; 83].

Because the coordination of many independently-controlled processors is so difficult, another option is to retreat from autonomous processors and have each processor apply a *common* instruction to its own local data. Because many data are massaged by each instruction, one can get considerable *data parallelism*, which can be spectacular for some problems. NASA's MPP [172] and the Connection Machine [93] are examples of these so-called Single-Instruction-stream, Multiple-Data-stream (SIMD) machines (Flynn's taxonomy [67]).

The promoters of INMOS transputers do not avoid the complexity of autonomous parallel operations; instead, they try to tame it with a clear theoretical model (Hoare's communicating sequential processes, embodied in the "occam" language [97]) and hot "transputer" silicon with firmware communication primitives to make the model viable [210]. Wired-together transputers are an example of a Multiple-Instruction-stream, Multiple-Data-stream (MIMD) approach.

These and many other techniques have earned multiprocessors a useful niche in today's computing scene. But it is worth asking: What would we *really* like to see in a multiprocessor to solve one problem faster? I suggest the following characteristics.

- It would be very fast compared to its same-technology sequential contemporaries, assuming enough parallelism to keep it busy.
- It could be applied to many, if not all, computing tasks.
- Programming the machine would include no extra hardships compared to programming a sequential computer. Presumably, there would no

longer be "sequential programming" and "parallel programming"—just "programming."

- It would be indefinitely *scalable*; one could keep adding processors to the machine with good results—either more speed on the same problems or the ability to solve bigger problems.

Let us return to programming, again. A radical solution to the software crisis is the technique of *functional programming*. A program is an ordinary mathematical function that yields an "answer" when applied to "input data." The most notable casualty of functional programming is the assignment statement of traditional "imperative" languages. Other features include:

- Functional programs deal only with *values*, not with the memory locations that happen to hold those values. John Backus describes current programming as figuring out what is to be done *and* preparing a "storage plan" to decide what location holds what value at which time [16]. Much of the code in an imperative program micro-manages the storage plan.

- *Lazy evaluation* ensures that nothing is evaluated unnecessarily. Lazy evaluation lets programs use infinite data structures; for example, in

  (first_three (all_primes_from 1)),

  all_primes_from would begin generating an infinite list of prime numbers, just enough for first_three to select the first three elements. (ML, used in this dissertation, is a functional language that does *not* use lazy evaluation.)

- Functional programs often use *higher-order functions*, those that take other functions as arguments or return them as results. The "compose" function, as in $f \circ g$, takes functions f and g as arguments.

- In his article "Why Functional Programming Matters," Hughes argues that lazy evaluation and higher-order functions make it easier to build kits of re-usable, mix-and-match program parts from which "modular" programs may be synthesized [106].

- Functional programs are much easier to reason about mathematically than imperative programs. For example, "linear" recursive functions may be transformed such that they can be encoded as efficient while-loops [15; 88]. Multiple passes over list-structures can often be reduced to fewer passes, with absolute certainty that the program's semantics are unchanged. Field and Harrison's text provides a good survey of approaches to "program transformation" [66].

- Functional programs are typically shorter than their imperative counterparts, and they can be dramatically clearer. (I hope the programs in this dissertation vindicate this viewpoint!)

There are several camps in the functional programming community. The "lazy purists" are purists because they eschew all mathematically-opaque language features and lazy because they insist on lazy evaluation (and the programming style it makes possible); this camp is in the ascendant. Miranda[1] and Haskell are lazy functional languages. The "Backus purists" also avoid "impure" features, but they follow John Backus in advocating "function-level thinking," a constrained use of higher-order functions, and an eager evaluation strategy (evaluate arguments *before* applying a function). FL is the most recent function-level language [17]. The "impurists" make up the largest camp; they allow imperative features but discourage their use. LISP, Scheme, and ML are representative languages. In this dissertation, I take the lazy purists' demands to heart, using techniques learned at my home base in the Backus camp, and coding my sample implementations in an impure language.

At the core of lazy functional languages is a formal system called the *λ-calculus*, which was developed by Alonzo Church [46]. To do lazy evaluation, the rules of the calculus must be applied in the so-called *normal order* (or a closely-related order). Chapter 2 introduces the normal-order evaluation of the λ-calculus.

Reduction, generally speaking, is an approach to computing in which program, data, and "state" are represented together in some structure, and computation proceeds by applying *reduction rules* to make incremental changes in the structure. For example, the paper-and-pencil arithmetic expression $(4+9)\times(3-7)$ includes data (the numbers 3, 4, 7, and 9) and program (the operator symbols $+$, $-$, and $\times$, plus their ordering with parentheses), and it could include other information (annotations, perhaps, to further constrain evaluation order for numerical error-control reasons). Applying the rules of arithmetic, the expression *reduces to* -52, in three reduction steps:

[1]Miranda is a trademark of Research Software Ltd.

$\rightarrow 13 \times (3-7)$, $\rightarrow 13 \times -4$, and $\rightarrow -52$. Reduction machines stand in contrast to *fixed-program* or *reentrant* machines, in which the computing instructions are segregated from program data and are not modified during program execution [209].

An expression in the λ-calculus is represented naturally by its *parse tree*, with the λ-calculus reduction rules causing changes to the tree; this is how a λ-calculus computation proceeds on a blackboard, for example. Unfortunately, normal-order computations may lead to *exponential* growth in both tree size and number of reductions to be done (Section 2.9). For this reason, the earliest computational mechanisms based on reduction of the λ-calculus (e.g., the SECD Machine [130]) used applicative order. Wadsworth's great contribution was to show that normal-order reduction was more practical if λ-calculus expressions were represented as *graphs*, with the rules of the λ-calculus carried out as changes to the graphs [201].

The main feature of graph reduction, roughly speaking, is the sharing of common subexpressions; it eliminates the exponential-growth problems and provides a parsimonious program representation. Furthermore, if an expression is shared, then the result of its evaluation will also be shared. Section 2.9 describes the space problems of standard λ-calculus expressions; Chapter 3 gives a full introduction to graph reduction, presenting a complete λ-calculus interpreter.

The prospects for multiprocessors and functional programming are linked. Historically, the implementations of lazy functional languages on von Neumann computers have been grossly inefficient when compared with traditional imperative languages. Recent implementations do much better; Peyton Jones's book covers the state of that art [165]. Still, one may argue that an imperative language (Backus: "von Neumann language" [14]) will always beat a functional language on von Neumann machines because the former is only the flimsiest disguise for the underlying stored-program uniprocessor machine. The functional language is a ballerina at an imperative square dance. A multiprocessor of appropriate design could better serve the functional language's requirements.

A functional approach offers great potential benefit to parallel computing. The powerful first Church-Rosser theorem (for the λ-calculus) allows great latitude in the order in which function applications are undertaken, guaranteeing that the results will be the same in every case. A parallel implementation is free to do the applications simultaneously. This no-intervention-needed parallelism is usually called *implicit parallelism*, and it is very important for scalable machines: one cannot expect a programmer to pre-plan the execution of hundreds of thousands of independent processing tasks "by hand." Consequently, functional programming—which proponents claim is superior anyway—holds the prospect for significant parallelism without any extra effort from programmers.

To design and build a successful multiprocessor that supports lazy functional programming is a truly monumental task, and it has not been done yet. Several parallel computers to support lazy functional programming have been designed, including Rediflow [114; 116], ALICE [56; 88; 49], and the Dutch Parallel Reduction Machine [23; 91] (Section 3.5 has the details). I will focus on their common design decision of a computational model[2] of *parallel graph reduction*, in which the many processing entities concurrently twiddle with a graph that represents program and data. Why did they all choose graph reduction? What did they gain? What implications did this choice have for their architectures? What were the eventual costs of the decision? These questions have not been examined carefully enough.

The obvious benefits of graph reduction stem from its sharing properties. Some of the imposed constraints are nearly as obvious. First, the program graph must be in a global store. As execution proceeds and the graph becomes more tangled, a node may have an outgoing edge pointing to *any* other node in the graph: the hardware must allow for this possibility. Second, in a distributed implementation, it becomes much harder to ensure that adjoining nodes in the graph will be in physically close-together hardware units. Third, it is difficult to move graph nodes around without leaving dangling pointers; re-shuffling nodes to improve locality is practically impossible.

Since I have been at the University of North Carolina, Chapel Hill, I have been privileged to work on the design team for the FFP Machine (FFPM), a highly parallel multiprocessor that directly supports Backus's FFP class of low-level functional languages, a suitable basis for "function-level" programming. (I urge you to read the introduction to the Machine in Section 5.1 if you are not familiar with the design.) The FFPM makes a bold attempt to be a completely scalable design (up to millions of processors), to be applicable to problems with dynamic data structures and unpredictable execution patterns, to remove the exploitation of parallelism from the programmer's worries, and to provide fully-automatic, deadlock-free storage management.

Though the main effort has been to support FFP-like languages, the FFPM project has also studied how to support other languages; Section 5.1.1

[2]Dally and Wills say, "A model of computation is a set of abstractions that provides a programmer with a simplified view of a machine. A model typically provides abstractions for memory, operations, and sequencing" [54, page 19].

takes up this matter in more detail. What about lazy λ-calculus-based functional languages on an FFPM? Project folklore knew "it could be done;" Plaisted published a brief description of one method in one of his 1985 papers on FFPM extensions [171]. (Section 5.4 reviews previous work about implementing λ-calculus-like languages on an FFPM.)

What are the main issues for FFPM support of lazy functional languages? First, the FFPM has a hardwired innermost-first evaluation order, corresponding to applicative-order, eager reduction; normal-order reduction is leftmost-first. (Sections 2.4–2.5 introduces a variety of normal forms and evaluation orders.) Happily, the two can be reconciled; Section 5.1.3 describes the technique.

The second issue arises because an FFPM normally operates on a linear symbol-string representation of an expression's parse tree, just as people do with pencil and paper. In this way, any subexpression (subtree) is *local* to a contiguous segment of the symbol-string. Since most useful tree manipulations work entirely within a subtree, the corresponding FFPM operations can be defined to work on localized contiguous symbol-strings. This property fits beautifully with the requirements for a scalable machine design, which strongly mitigate against the sharing of system resources. (For example, four processors sharing a page table is OK, but 400,000 processors will find access slow.) The operating cycle of an FFPM reflects an aversion to sharing: close-together processors are partitioned into groups that hold an interesting subtree's worth of symbols, and that group tries to proceed on a reduction. Because the subtree has *all* information required for a reduction to proceed, the processor group has no need to share *any* resources with other groups. But, for many reductions to be going at once, information common to many reductions must be *copied* enough times so that each processor-group has a full set of information. This may seem wasteful (and it can be), but it allows potentially many computations to proceed *entirely* independently of each other. This is what it means to say an FFPM "favors copying" and "exploits locality."

Operations on a tree may be localized, but the tree's size will change and may shrink dramatically or grow arbitrarily large. To deal with this fluctuation, an FFPM provides automatic storage management in hardware, allowing program symbols to be inserted or deleted anywhere in a program string. Program execution is then an ongoing "cut and paste" re-arrangement of symbols (Révész, personal comment). This means there is no guarantee what symbol a given processor will hold on any given machine cycle. Therefore, the memory distributed across the many processors does not have addresses in the usual sense, and one processor can request something from another only by *value*; for example, "Will the processor holding index value 4 please send its program symbol?" There is no such thing as "the processor with address 4." This addressless property of memory in an FFPM—essential for storage management—makes it less than ideal for representing graphs, which are usually implemented as nodes with memory-cell-pointers for edges. One can *simulate* memory-cells-with-addresses in software (Magó's study of Paterson-Wegman unification is described in Section 5.1.11), but it is not a natural fit.

I have already mentioned that the normal-order evaluation may suffer exponential blow-ups unless sharing is done. One would therefore assume than an FFPM, disinclined to sharing, would make for a poor normal-order λ-calculus implementation. Perhaps... But there may be a way to go about sharing in an "addressless way." Can the useful features of an FFPM be brought to bear on the λ-calculus from a different angle?

Lazy functional language implementations without some form of sharing are completely impractical, so I will use sharing as my basic measure of success. Graph reduction is one way to achieve the desired sharing. On the other hand, the sharing of conventional graph reduction—nodes linked by pointers in a global addressable memory—is ill-matched to the desiderata for scalable highly-parallel computers. These machines strongly favor representations that preserve locality.

1.2 Thesis statement

This dissertation examines the implementation of lazy functional languages on highly-parallel computers by focusing on a restricted "archetypal" problem, the normal-order evaluation of the pure λ-calculus. This is what Wadsworth did in his original work on graph reduction [201]. Enough sharing to avoid the likely exponential blow-ups of naive non-graph reduction is required, and matching the maximal sharing of graph reduction is desirable.

This dissertation develops a system for normal-order evaluation of the λ-calculus that represents terms as trees instead of graphs (Chapter 4). Graph manipulations imply global pointers into a common store, a major impediment to implementation on highly-parallel computers. I compare my tree-reducing system (Chapter 4) to a graph reducer that does "lazy" copying of shared functions (Chapter 3). I claim:[3]

[3]It is impossible to make these claims as precise as I would prefer until the scaffolding of

> My tree-reducing interpreter manipulates terms-as-trees in a way isomorphic to a lazy-copying terms-as-graphs reducer, step for step. Because graph reduction is a correct implementation of the λ-calculus, the tree-reducing interpreter must be as well.

To consider worst-case time and space complexities, one must consider a reduction system in light of some computational model. To this end, I consider my interpreter implemented on an FFP Machine (Chapter 5); other architectures that support a linear program-representation and fast scan primitives would also work. I compare this with graph reduction on a conventional global-addressable-memory (GAM) machine, and I claim:

> The FFPM implementation of the tree-based reducer uses the same amount of space (within a constant factor) as conventional graph reduction on a GAM machine. Moreover, the FFPM implementation matches or improves on the time complexity of each part of a reduction step, with the exception of "last-instance relocations," a non-critical operation. (Section 5.3.2 reviews this obscure matter in painful detail.)

I believe the general approach suggested in this dissertation might well provide a viable base for highly-parallel computing systems to support lazy functional programming.

1.3 Dissertation organization

After an introduction to the λ-calculus in Chapter 2, the heart of the dissertation (Chapters 3–5) is a comparison of two interpreters for the pure λ-calculus. The first is a standard graph reducer (Chapter 3), and the second is my new "suspension-based" tree reducer (Chapter 4). I compare them for correctness (Section 4.7), space complexity (Section 5.3.1), and time complexity (Section 5.3.2). The beginning of Section 4.7 describes the strategy for the comparisons.

To compare space and time complexity, one must consider the interpreters in the context of some computational model. For graph reduction, I use a

the next chapters is in place. These claims are recapitulated in the Conclusions, Chapter 7, page 163.

conventional GAM machine; for the new interpreter, I consider its implementation on an FFPM (Chapter 5); that chapter concludes with a comparison of the two (Section 5.3).

Chapter 6 catalogs some ideas for extending the new interpreter that might be useful in turning this work into a practical parallel computing system. Chapter 7 presents my conclusions about this work.

I review previous and related work after topics have been introduced, at the end of the appropriate chapter or section. For example, I survey graph-reduction architectures for functional programming in Section 3.5, just after introducing my graph reducer.

Chapter 2

The λ-calculus

What brings a parallel processing enthusiast into the jungles of the lambda calculus, a harsh and hostile territory replete with expressions so ugly that only a mathematician could love them?

— Almasi and Gottlieb (1989).

This chapter introduces a formalism called the *λ-calculus*,[1] setting the stage for the interpreters in Chapters 3 and 4. The λ-calculus underlies all lazy functional languages and captures their essential properties.

Alonzo Church invented the λ-calculus as a precise notation to study functions [46]. In the λ-calculus, a function is viewed as a *rule* that converts arguments to values, rather than as a set of (argument, value) ordered pairs. It is this rule-oriented view of functions that brings out their computational aspects [18]. John McCarthy developed LISP, the first widely-used programming language influenced by the λ-calculus [149]. Peter Landin showed the connection to other programming languages (Algol 60, in that case) [131] and went on to suggest that future languages would be "syntactic sugarings" of the λ-calculus [132]. Dana Scott and Christopher Strachey did the crucial work to provide a well-founded denotational semantics for the λ-calculus and, by implication, the programming languages based on it [182]. Wadsworth's development of graph reduction (Chapter 3) was a major step forward for the implementation of λ-calculus-based functional languages [201]. Backus's

[1]Strictly speaking, the untyped pure λK calculus; there are other variants.

1977 Turing Award lecture [14] widened interest in functional programming, and research has continued unabated since then.

Although the λ-calculus is spare and simple, the results *about* it are profound and sometimes taxing. For a complete treatment, Barendregt [18] is the standard reference. Hindley and Seldin's book [96] is a more accessible treatment; most functional programming texts devote at least one chapter to the λ-calculus.

2.1 Syntax

Stated in programming-language terms, the λ-calculus is a systematic way of describing functions and their application to arguments. Let us begin with its syntactic elements.

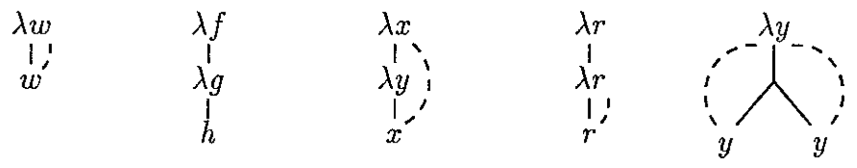
Well-formed expressions in the λ-calculus are called *terms* (or *λ-terms*). The simplest terms are *variables* (shown by a lower-case letter): x, k, y, or b, for example.

The second kind of term is the *abstraction* (or *λ-abstraction*): this is how functions are defined. A λ-abstraction has the form $\lambda v.\{T\}$, where v is a variable and T is any term (capital letters denote arbitrary terms). The unconventional braces to delimit a λ-abstraction are necessary later in this dissertation, to avoid implicit scope rules. A better notation, using only braces, might be $\{_v T\}$; however, I keep the λ symbol, etc., because readers expect to see λ's in the λ-calculus.

A λ-abstraction defines a function with a formal parameter v and a *body* T; the body specifies what the function "returns." For example, $\lambda x.\{x\}$ is a function of x that returns whatever is passed to it—it is the identity function. Another example is $\lambda r.\{s\}$, a function that returns s no matter what is passed to it—it is the constant function s. As one would expect, the specific name of the formal parameter x is irrelevant. $\lambda x.\{x\}$ and $\lambda y.\{y\}$ are the same function.

In the term $\lambda x.\{\lambda y.\{x\}\}$, the *subterm* $\lambda y.\{x\}$ is the body of the whole term, and the variable x is *bound* to the first λ. That λx is called the *binder* of x; a variable has at most one binder. Only λ's can be binders. Conversely, the variable x is a *bound variable* of the λx; one λx may have many bound variables, all denoted by the name x. In a term $\lambda c.\{D\}$, the bound variables of λc must fall within D. D—or the term enclosed by the braces $\{\ \}$—is the *scope* of λc.

A variable a is bound to the innermost λa in whose scope it falls, as in



$\lambda w.\{w\}$ $\lambda f.\{\lambda g.\{h\}\}$ $\lambda x.\{\lambda y.\{x\}\}$ $\lambda r.\{\lambda r.\{r\}\}$ $\lambda y.\{(y\ y)\}$

Figure 2.1: λ-abstractions

lexically-scoped programming languages. In $\lambda r.\{\lambda r.\{r\}\}$, the variable r is bound to the rightmost λr. If, in a term R, there is no λa, then occurrences of a variable a are not bound, but *free*. Note that x is bound in $\lambda x.\{\lambda y.\{x\}\}$ but is free in the subterm $\lambda y.\{x\}$. A variable that has no binder anywhere is free at the top level; such variables are *constants* .

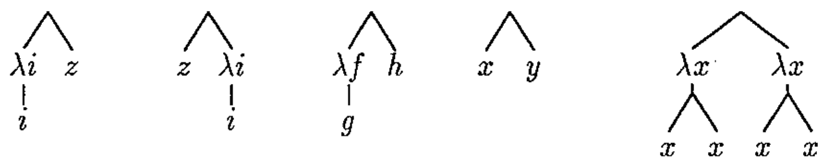
Figure 2.1 shows some examples of λ-abstractions; h is a constant; w, x, r, and the y's are variables, bound in the λ-terms shown. The parse trees—or λ-*trees*—for the terms are shown above their text representations (dashed arcs show bindings of variables to binders, if they exist). I find the trees easier to understand; written-out terms of more than, say, seven symbols make my eyes glaze over.

Variable names and bindings can be confusing when a λ-term contains two variables named x with different binders (for example). To substitute for one x but not the other is tricky, and the procedure cannot be automated efficiently. I avoid this problem by changing to a *name-free* λ-*calculus* in Section 2.6; meanwhile, I restrict myself to λ-terms in which the problem does not arise.

The final construct of the λ-calculus is an *application* (or λ-*application*), of the form $(F\ X)$—the function term F is applied to the argument term X. For the term $(F\ X)$, F is the *rator* and X is the *rand*; the terms come from Landin [130] and are short for "operator" and "operand." Figure 2.2 shows seven λ-applications (shown in the λ-trees by unlabeled two-child nodes).

Tree representations prompt some useful definitions. The *binding path* (of a variable) is the (unique) path up the tree from the variable to its binder. The *root path* of any syntactic element (variable, λ-abstraction, or λ-application) is the path from the element to the root of the tree representing the whole term.

Some of the λ-applications in Figure 2.2 match our intuitions about applying a function to an argument. For example, applying the identity function $\lambda i.\{i\}$ to z should (and does) yield z. One surprise of the λ-calculus, however,



$(\lambda i.\{i\}\ z)$ $(z\ \lambda i.\{i\})$ $(\lambda f.\{g\}\ h)$ $(x\ y)$ $(\lambda x.\{(x\ x)\}\ \lambda x.\{(x\ x)\})$

Figure 2.2: λ-applications

is that the "backwards" term $(z\ \lambda i.\{i\})$ is equally well-formed.

Summarizing, a Backus-Naur-style grammar for well-formed λ-terms is:

$$\begin{array}{lcl} <\!\mathit{term}\!> & ::= & (<\!\mathit{term}\!>\ <\!\mathit{term}\!>) \\ & | & \lambda\ <\!\mathit{variable}\!>\ .\{<\!\mathit{term}\!>\} \\ & | & <\!\mathit{variable}\!> \end{array}$$

2.2 Computing with the λ-calculus

How does one compute with the λ-calculus? Intuition remains a reasonable guide: We apply a "program" rator to an "input-data" rand and hope that an "answer" will eventually be computed. (Because all computable functions can be expressed in the λ-calculus, the Halting Problem precludes any assurance of termination).

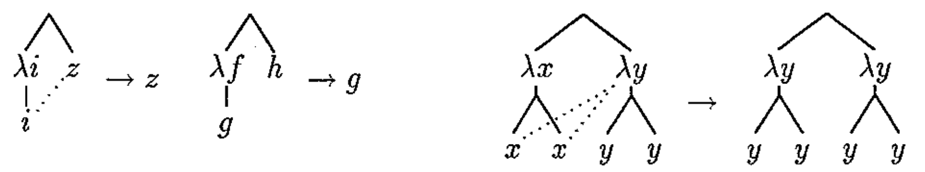
The fundamental operation of the λ-calculus is *β-reduction*. It defines what happens when a function-rator $\lambda x.\{M\}$ is applied to a rand N. The process is as simple as can be—N is textually *substituted* for every variable bound by λx in M. Or, as it is usually expressed, N is substituted for every free occurrence of x in M. In symbols, a β-reduction is written as

$$(\lambda x.\{M\}\ N) \to M[x := N].$$

In a β-reduction, the substitution for the variable is the main effort (not the other minor adjustments to symbols in the term). Figure 2.3 shows some β-reductions (dotted lines show the substitutions and point downward).

Consider Figure 2.3c. The β-reduction substitutes $\lambda y.\{(y\ y)\}$ for every free occurrence of x in $(x\ x)$—both of them. This example also shows that a reduction does not necessarily produce something shorter; the left and right terms are the same—suggesting a nonterminating sequence of reductions.

A term of the form $(\lambda x.\{M\}\ N)$ is a *β-redex* (reducible expression). Reducing one redex is called a *β-reduction step* (or just a "step"). A term



$(\lambda i.\{i\}\ z) \to z$ $(\lambda f.\{g\}\ h) \to g$ $(\lambda x.\{(x\ x)\}\lambda y.\{(y\ y)\}) \to (\lambda y.\{(y\ y)\}\lambda y.\{(y\ y)\})$

(a) (b) (c)

Figure 2.3: β-reductions

T_1 *β-reduces to* a term T_2 if T_2 can be obtained from T_1 by a finite sequence of zero or more steps.

Though not done often, the β-rule may be invoked in reverse:

$$(\lambda x.\{M\}\ N) \leftarrow M[x := N].$$

This is a *β-expansion step*. To express that the β-rule may be used "in both directions," one speaks of *β-conversion*.

2.3 Other λ-calculus reduction rules

The pure λ-calculus has more fundamental rules for manipulating λ-terms. This section says why I do not pay them much attention.

The *α-rule* (or its use, *α-conversion*) renames variables to avoid name clashes. I skirt this issue by using either a name-free calculus (Section 2.6; Chapter 4) or backpointers (Chapter 3).

The *η-rule* (or its use, called an *η-reduction* step) is

$$\lambda x.\{(Mx)\} \to M.$$

There must be no free occurrences of x in M. The η-rule is needed for extensional equivalence. It is useful in compile-time transformations, but it is not needed for "computing." Peyton Jones's book about implementing functional languages gives further details [165, pages 19–20].

As with the β-rule, the η-rule may be used in reverse: it is then *η-expansion*. Using the rule both ways is *η-conversion*. When not qualified, "reduction," "expansion," and "conversion" refer to the β-rule.

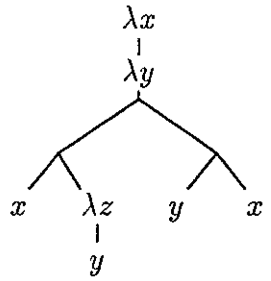


Figure 2.4: A λ-term in β-normal form

2.4 β-normal form and normal-order reduction

I have introduced the three syntactic elements of the λ-calculus—variables, λ-abstractions, and λ-applications—and the main operation, β-reduction, which has substitution as its main component. What does one *do* with it?

An obvious possibility is to do reduction steps until there are no more β-redexes. A term that contains no β-redexes is in *β-normal form* (BNF). Figure 2.4 shows a term in BNF.

A term T in BNF is *unique* (up to renaming of variables), in that no other term in BNF can be reduced to it. Since a term in BNF is what is left when computation is finished—an answer, in some sense—its uniqueness is exceedingly important.

What of a term that contains redexes than cannot be removed by any sequence of reduction steps? Figure 2.3c is an example. Its reduction is *non-terminating*, so it has *no* BNF.

Finally, consider a term T with many redexes that has a BNF T_{BNF}. Will we reach T_{BNF}, no matter what order we do β-reductions? No—we could end up down a blind alley of non-termination. For example, in the term

$$(\lambda q.\{r\}\ (\lambda s.\{(s\ s)\}\ \lambda s.\{(s\ s)\}))$$

if we always choose the rightmost redex, we will not reach BNF, whereas the other (left) redex yields r in one step.

Fortunately, there does exist an *evaluation order*—a pre-defined order in which β-reductions should be done—that will yield a term's BNF if it exists; this is the second Church Rosser theorem. It is called the *normal order*; evaluation using this order is called *normal-order evaluation.* When terms are written as linear text, the *leftmost* β-redex should be reduced in each step. The equivalent λ-tree rule is to choose the first redex reached by a preorder walk of the tree from the root.

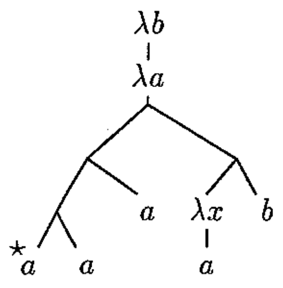


Figure 2.5: A λ-term in head-normal form

Normal-order evaluation is lazy—it reduces a β-redex only if necessary. This property allows programming with infinite data structures, a feature that the lazy functional programming community insists upon.

2.5 Other normal forms and evaluation orders

Evaluating to BNF, while conceptually simple and theoretically satisfying, traditionally leads to inefficient implementations, mainly because free variables in λ-abstractions (that are rators) preclude their effective compilation (Peyton Jones illustrates the problem in his book [165, pages 221–222]). Therefore, one might evaluate terms to another normal form deemed sufficient grounds to stop reducing.

When the λ-calculus is given some semantics (a matter I am ignoring), terms without a BNF are considered "meaningless." *Head-normal form* (HNF) is a less restrictive normal form that retains this property: a λ-term without a HNF is also "meaningless" so evaluating to HNF is just as good. A λ-term is in HNF *iff* it is of the form

$$\lambda x_1.\{\lambda x_2.\{\ldots\lambda x_n.\{(\cdots((v\ M_1)\ M_2)\cdots M_m)\}\}\}$$

where $n, m \geq 0$, and v is a variable [165, page 199]. Figure 2.5 shows a term in HNF. In λ-tree terms, if you throw away a term's top-level λ's (λb and λa in Figure 2.5), then walk along the left "spine" of application nodes (three of them in Figure 2.5), the first *non*-application node is at the *head*, shown by a $\star$ in Figure 2.5. If the head node is not a λ-abstraction (i.e., it is a variable), then the term is in HNF. HNF is a less constraining normal form: a term in BNF is in HNF, but a term in HNF is not necessarily in BNF; the example in Figure 2.5 still has a redex in it. If that redex led to a non-terminating sequence of reductions, then the term would *not have* a BNF, yet it is in HNF. Wadsworth invented HNF [201]; Barendregt also discusses it [18, page 41]. Berkling wrote an interesting paper about evaluating to HNF [25].

HNF satisfies purists, but it can still have free variables in redexes. For this, *weak head-normal form* (WHNF) is required. As I use it, weak reduction means "redexes inside λ-abstractions do not count." So, even if Figure 2.5 had a redex at its $\star$'d head position, it would be in WHNF because it is inside the top-level abstraction $\lambda b.\{\dots\}$. The effect of reducing only to WHNF is that one never has to contend with free variables in a redex (excluding constants) [165, page 198]. A term in HNF is also in WHNF, but a term in WHNF may not have a HNF.

BNF, HNF, and WHNF are the most common normal forms; WHNF is the most popular for lazy functional-language implementations. (Peyton Jones's book on implementation has a lot to say about WHNF [165].)

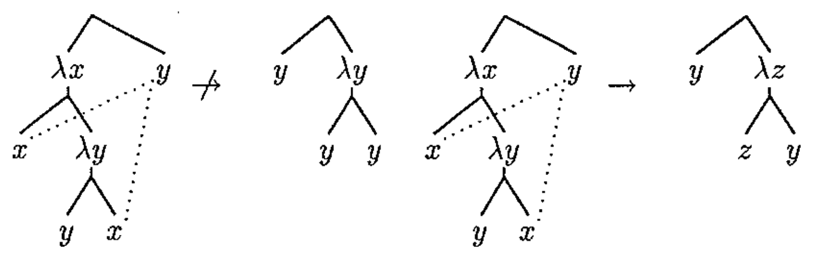
I must introduce more forms that will be needed later on. *Weak β-normal form* is to BNF as WHNF is to HNF: redexes are not reduced inside λ-abstractions, top-level ones excluded.[2] *Lambda form* (LF) means that the term's λ-abstractions that cannot possibly be rators of any redex are in BNF. (This truly obscure definition is used in the traditional eval-apply interpreter for BNF, eval_BNF (page 25).) *Root-lambda form* (RLF) is a still-more-obscure form used the interpreter in Chapter 4, and I defer its definition until then.

BNF is the only normal form listed that is unique, provided it exists. A term T may have two distinct β-convertible variants, $T_1 \leftrightarrow T_2$, both in the normal form. This can happen because of a redex in an "uninteresting" subterm, e.g., for HNF, not in head position.

Various evaluation orders can be used to reach a particular normal form. An order that is certain to produce the normal form for a term if it exists is a *safe order* for that form. Normal order is safe for all normal forms mentioned above.

An *unsafe order* is an evaluation order that may fail to find a term's normal form in some cases (presumably rare). The most common unsafe order is applicative order, in which the rand and rator of a redex are evaluated *before* doing the β-reduction; it is one form of *eager evaluation.* Applicative order is practical—the entire LISP community uses it—but it does preclude computing with infinite structures. This disqualifies it for lazy functional programming.

[2]The exclusion is one of convenience and has no deeper significance.



(a) wrong! (b) right

Figure 2.6: How to capture a variable in one easy lesson

In this dissertation, I use normal-order evaluation to BNF, or WBNF when BNF poses a major implementation hurdle. Both are adequate for lazy functional programming.

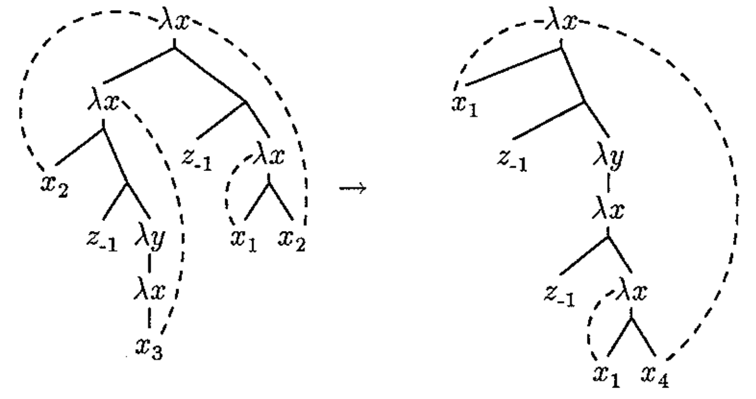
2.6 The name-free λ-calculus

Names for variables in the λ-calculus are convenient for the reader but complicate some definitions, most notably the precise definition of $M[x := N]$, substitution of N for free occurrences of x in M. The classic difficulty is *name capture*; the reduction in Figure 2.6 illustrates the problem (dotted lines highlight the substitutions): Unless the λy in the left-hand side is renamed (along with the variable y bound to it), it will capture the (unrelated) y being substituted for x. Figure 2.6b shows the same reduction with the necessary renaming.

A common solution to the naming problem is to represent each variable by its *binding index*, the number of binders on a variable's binding path. This is easiest to see on a λ-tree: start at a variable and walk toward the root, counting binders (λ's). Note that a variable with no λ's between it and its binder will have a binding index of 1; others might define it to be 0.

De Bruijn [59] invented binding indices, and many people call them "de Bruijn numbers;" Berkling [27; 32] developed a closely-related scheme independently; the term "binding index" is his [25].

To convert all variables to binding indices, constants must get indices, too. I give them distinct negative binding indices. Different instances of the same constant will have the same negative index. Rules for the name-free λ-calculus must not change them. Because they do not change and convey little information, I usually write constants' binding indices as a $*$ subscript.



$$\lambda x.\{(\lambda x.\{(x_2\ (z_{-1}\ \lambda y.\{\lambda x.\{x_3\}\}))\}\ (z_{-1}\ \lambda x.\{(x_1\ x_2)\}))\} \rightarrow$$
$$\lambda x.\{(x_1\ (z_{-1}\ \lambda y.\{\lambda x.\{(z_{-1}\ \lambda x.\{(x_1\ x_4)\})\}\}))\}$$

Figure 2.7: Reduction with binding indices

Figure 2.7 shows a name-free reduction; I have added the binding indices as subscripts to the variable names. Dashed arcs show the bindings of bound variables to binders (λ's).

Once a term has had binding indices added, the variable names can be removed. This is often done and leads to a notation such as

$$\lambda.\{(\lambda.\{(2\ (\text{-}1\ \lambda.\{\lambda.\{3\}\}))\}\ (\text{-}1\ \lambda.\{(1\ 2)\}))\} \rightarrow \lambda.\{(1\ (\text{-}1\ \lambda.\{\lambda.\{(\text{-}1\ \lambda.\{(1\ 4)\})\}\}))\}$$

Readability has not improved,[3] so I prefer keeping a term's original variable names as mnemonic decoration, with the real information (binding indices) attached as subscripts, as in the example above. The rules of a name-free calculus operate *only* on the subscripts. The names are there only to make examples easier to understand. The following definition will be used later.

Definition 2.1 Two name-free λ-terms are *λ_t-equivalent* if they are identical (but the name decorations on the terms do not count, of course).

The predicate function plain_equivs (page 23) checks if two plain λ-terms are λ_t-equivalent. The t subscript suggests that it operates on terms as *λ-trees*.

A word on still other approaches to variable naming... Révész has an alternate calculus for which "brute-force" renaming works [174]. Staples

[3]We could make matters worse by removing the λ symbols, as in Chapter 5.

[191] and Boom [36] propose schemes in which extra information is pinned onto λ-abstractions to keep track of their bound variables. Appendix C of Barendregt's tome [18] discusses free and bound variables further.

A simple interpreter and the recurring example. We now have all the pieces to put together a simple normal-order interpreter for the name-free λ-calculus, written in ML (Appendix A gives some background information about ML). The function onestepT : Term $\rightarrow$ (bool,Term) (page 23) tries to do one β-reduction step. If it cannot, it returns false and the λ-term is in WBNF; otherwise, it returns true and the reduced-one-step λ-term. A top-level routine (not shown) repeatedly calls onestepT until it returns false. Figure 2.8 shows all the steps in the reduction of a recurring example that will be repeated for all interpreters in this dissertation.

The ML function eval_BNF : Term $\rightarrow$ Term (page 25) is a traditional encoding of an interpreter that reduces to BNF. The auxiliary function eval_LF ensures that a λ-abstraction used as a rator of a redex is not evaluated before reduction; other abstractions are fully reduced. Eval_WBNF is also shown; eval_WBNF and repeated calls to onestepT produce the same result, of course!

2.7 Combinators

An alternate solution to the name-capture problem is remove free variables altogether by converting λ-terms to *combinators* [52; 53; 164]. A combinator is simply a λ-term with no free variables except constants; $\lambda x.\{x\}$ and $(\lambda x.\{\lambda y.\{(x\ y)\}\}\ \lambda z.\{z\})$ are examples.

Any λ-term may be converted to a combinator-term built from a fixed set of combinators. The minimal set of building-blocks is the SK combinators.

$$\begin{aligned} S &= \lambda x.\{\lambda y.\{\lambda z.\{((x_3\ z_1)(y_2\ z_1))\}\}\} \\ K &= \lambda x.\{\lambda y.\{x_2\}\} \end{aligned}$$

Larger base sets of combinators may be used for more space-efficient encodings (e.g., Turner's combinators [199]). An alternate method is to use an even more specialized kind of λ-term called *supercombinators* [105]. A supercombinator is a combinator in which all inner λ-abstractions are *also* supercombinators. Supercombinators may have several arguments, and "multi-argument" reduction must be used to avoid intermediate terms that have free variables. Sets of supercombinators derived from specific λ-terms have some advantages for implementation over fixed-combinator sets.

```
(* plain_equivs : Term → Term → bool.
   Compares two TermTs and returns true if they are identical except for decorative
   names and variable marks; otherwise, returns false.
*)
exception unexpected_suspension_error

fun plain_equivs (App(M1,N1)) (App(M2,N2)) =
      (plain_equivs M1 M2) andalso (plain_equivs N1 N2)
  | plain_equivs (Lam(B1,_)) (Lam(B2,_))        = plain_equivs B1 B2
  | plain_equivs (Var(bi1,_,_)) (Var(bi2,_,_))  = (bi1 = bi2)
  | plain_equivs (Sus(_,_,_)) (Sus(_,_,_))      = raise unexpected_suspension_error
  | plain_equivs other1 other2                  = false

(* onestepT : Term → (bool,Term).
   Uses std_subst (page 172) and incr_free_vars1 (page 171).
   Find the first redex in T and reduce it (tree reduction); report whether or not a
   redex was done (and return the new λ-term).
   All of the code except the first clause implements a preorder walk of the term
   looking for a redex (an App node with a Lam rator).
   When an App(Lam(B,...),N,...) is found, real work begins. The main effort is sub-
   stituting N for bound variables of the Lam(...); the standard routine std_subst does it.
   The two calls to incr_free_vars1 adjust the binding indices.
*)
fun onestepT (App(Lam(B, n), N)) = (* redex *)
     (true, incr_free_vars1 ~1 (std_subst (incr_free_vars1 1 N) B))
  | onestepT (App(M, N)) = (* rator not a lambda *)
     (* do preorder walk *)
     let val (done_in_M, M') = onestepT M in
          if done_in_M then
               (true, App(M', N))
          else let val (done_in_N, N') = onestepT N
            in (done_in_N, App(M, N')) end
     end
(* — onestepT (Lam(B, n)) = (? if we were going to β-normal form...?)
      let val (done_in_B, B') = onestepT B
      in (done_in_B, Lam(B', n)) end
*)
  | onestepT other = (false, other) (* variable or abstraction [to WBNF] *)
```

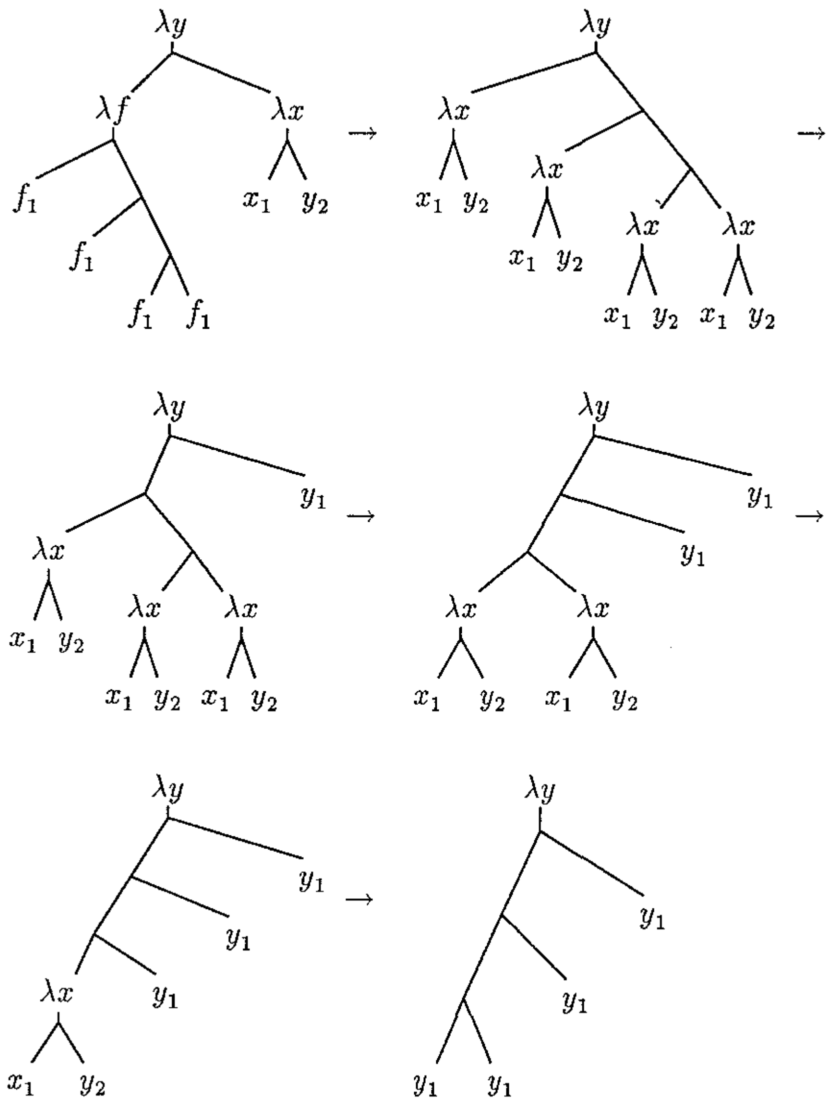


Figure 2.8: The recurring example: five tree reduction steps

```
(* eval_BNF : Term → Term.
   Uses std_subst (page 172), incr_free_vars1 (page 171), and eval_LF.
   This is the traditional normal-order interpreter, following Wadsworth [201,
   page 181]; the same thing is in Arvind et al. [9, page 5.3].
   eval_BNF evaluates a Term to β-normal form. The "helper" function eval_LF
   evaluates to lambda form, with no reduction inside a λ-abstraction that might
   become a redex-rator.
   eval_WBNF : Term → Term  evaluates a Term to weak β-normal form; redexes inside
   λ-abstractions are allowed to live.
   std_subst does substitution, and incr_free_vars1 keeps binding indices in order.
*)
fun eval_BNF (App(M,N)) =
    let val M' = eval_LF M in case M'
        of Lam(B, n) ⇒
            eval_BNF (incr_free_vars1 ~1 (std_subst (incr_free_vars1 1 N) B))
         | _ ⇒
            App(M', eval_BNF N)
    end
  | eval_BNF (Lam(B,n)) = Lam(eval_BNF B, n)
  | eval_BNF a_variable = a_variable

and eval_LF (App(M,N)) =
    let val M' = eval_LF M in case M'
        of Lam(B, n) ⇒
            eval_LF (incr_free_vars1 ~1 (std_subst (incr_free_vars1 1 N) B))
         | _ ⇒
            App(M', eval_BNF N)
    end
  | eval_LF (Lam(B,n)) = Lam(B, n) (* don't eval body! *)
  | eval_LF a_variable = a_variable

(* to weak normal form... *)
fun eval_WBNF (App(M,N)) =
    let val M' = eval_WBNF M
    in (case M'
          of Lam(B,n) ⇒ (incr_free_vars1 ~1 (std_subst (incr_free_vars1 1 N) B))
           | _        ⇒ App(M', eval_WBNF N)
    ) end
  | eval_WBNF other = other (* abstraction or variable *)
```

Combinators have no free variables, so they are pure rewrite rules and require no "context" for their evaluation. This property makes the nature of a combinator-based interpreter quite different, and the strategies used diverge widely from those used for evaluating the pure λ-calculus. I will have little more to say about combinators.

2.8 The practical use of the λ-calculus

The pure λ-calculus is not a practical medium for computation; for example, it has no numbers and no arithmetic. The theoretically-minded will be happy to know that these can be represented in the pure λ-calculus. But, even then, the pure λ-calculus remains wildly impractical, so the designer of a λ-calculus-based language always adds numbers, arithmetic and other primitives. Moreover, theoreticians may add different symbols or restrictions to the formal system for their own nefarious purposes. So, there are many λ-calculus and λ-calculus-derived systems, each contrived for a different purpose.

From a FORTRAN[4] programmer's perspective, a normal-order interpreter of the λ-calculus (or a practical variant) is inefficient: it is too slow, and it uses too much memory. The impediments run deep: substitution of full generality, as in β-reduction, is not a bounded operation, there is no really efficient representation for higher-order functions, and the bookkeeping overhead needed to track variables' freeness can be considerable. This inefficiency has been tackled in many ways, including:

- Make infrequent use of the parts that are "inefficient." LISP, the first programming language based on the λ-calculus, also has full imperative features that readily compile to good global-addressable-memory (GAM) machine code. Most real LISP programs are not written in a functional style. Also, LISP's applicative-order evaluation is unsafe.

- Use a different evaluation order, aim for a different normal form, and provide many primitive operations; in short, soup up the base language.

- At compile-time, transform the initial λ-terms into something more amenable to efficient execution on a GAM machine (e.g., to supercombinators [105]).

[4]Phil Wadler, visiting UNC in the fall of 1984, paraphrased: "Let's call all languages with assignment 'FORTRAN'."

- Use hints from the programmer to improve the efficacy of compilation.
- Improve the interpreter's basic model of computation, upgrade its algorithms, or augment its realization (e.g., throw hardware at it).

2.9 The necessity of sharing for normal-order evaluation

I want an interpreter for the normal-order evaluation of the λ-calculus that does not depend on representing λ-terms as graphs. Graphs are normally used because they can represent *shared* terms easily (see Chapter 3). This section explains why the sharing is necessary.

What is "sharing?" It means that *one* instance of a term S is made to stand for *many* occurrences of the term. For example, in the arithmetic expression

$$x + x + x, \ x = 2 \times 2,$$

the product 2×2 is "shared;" the references to it, x, are generally called "pointers". Whereas the shared instance of a term may be arbitrarily large, the pointers to it are constant-sized.[5]

If the size of a pointer and the shared term it points to are the same (within a constant factor), that is *trivial sharing*, because it does not save any space. In the λ-calculus, sharing a variable is trivial.

Non-trivial sharing of the kind just described is *space sharing*; memory is conserved. A second form of sharing is *computation sharing*, in which reduction-steps are conserved:[6] when a space-shared term S is reduced to S', all the pointers to S will (by some magic) indicate S'. If one of those pointers is followed later, the reduction $S \rightarrow S'$ need not be re-done.

As Section 3.3 will make clear, the λ-calculus requires some copying of λ-terms, even for graph reduction. The copying that must be done for correctness' sake is *necessary copying*; other copying is *unnecessary*. Unnecessary copying done willfully in hope of some benefit (e.g., speeding things up) is *speculative*; one cannot determine the necessity of copying in advance.

Why is sharing practically required for the normal-order λ-calculus? Consider, informally, a simple, normal-order evaluation without sharing versus an

[5]Strictly speaking, a pointer into an arbitrarily-large address-space is also unboundedly big; however, I am following the computer-science practice of believing that any pointer can fit into 32 or 64 or 128... bits.

[6]I am following the terminology of Arvind et al. [9].

applicative-order one (I follow Wozencraft and Evans's notes for MIT course 6.231 [213, pages 3.2-32–3.2-33]). Intuitively, applicative order reduces the redexes at the bottom of the λ-tree and passes the results upward to the next level of reductions. Two good things *may* happen. First—and this is the weaker argument—reductions often produce smaller λ-terms, using less space when passed upward and copied by higher-up redexes. Second, a λ-term being substituted never contains a redex, so there is *no proliferation of unevaluated redexes*. The problem with applicative order is that some of those bottom-up reductions are unnecessary (their results will be thrown away later) and, in the worst case, non-terminating—which is why applicative evaluation is unsafe.

Normal-order reduction, by contrast, only commits to reductions that are *certain* to be needed in getting to BNF. (Determining "neededness" in general is undecidable; Barendregt et al. discuss some approaches to this matter [19].) Since the leftmost redex is always needed, normal-order evaluation reduces it at each step. Meanwhile, normal-order reductions may make many copies of *unevaluated* λ-terms. These terms are likely to be larger than their β-reduced equivalents; moreover, copying them can increase the number of redexes in the whole λ-term.

This argument is informal, because cases can be concocted to show either applicative or normal order superior. However, common cases of function composition—extremely important in practice—get normal-order-evaluation-with-copying in trouble. Figure 2.9 shows an example in which a function f is composed with itself, $(\lambda f.\{(f\ (f\ (f\ (\lambda y.\{y_1\}\ z))))\}\ \lambda x.\{(x\ x)\})$; redexes are starred. Applicative order quickly determines the initial argument $[(\lambda y.\{y_1\}\ z) \rightarrow z]$, substitutes $\lambda x.\{(x_1\ x_1)\}$ for f, does the compositions right-to-left, bottom-to-top, and finishes the whole job in five steps, total. Normal-order evaluation, on the other hand, substitutes for f first, then does the compositions left-to-right, top-to-bottom, each time substituting the *whole* right part of the term, eventually making eight unevaluated copies of the initial argument, $(\lambda y.\{y_1\}\ z)$. The reduction takes sixteen steps.

If there were k uses of f in Figure 2.9 (instead of three) and there were n instances of x in the rand (instead of two), then applicative-order evaluation would reduce the λ-term in $2 + k$ steps. Normal-order would take roughly n^k steps. This *exponential blow-up*, both in space required and reductions to do, *will* arise in practical normal-order reductions; therefore, some sharing mechanism *must* be provided for any normal-order λ-calculus interpreter.

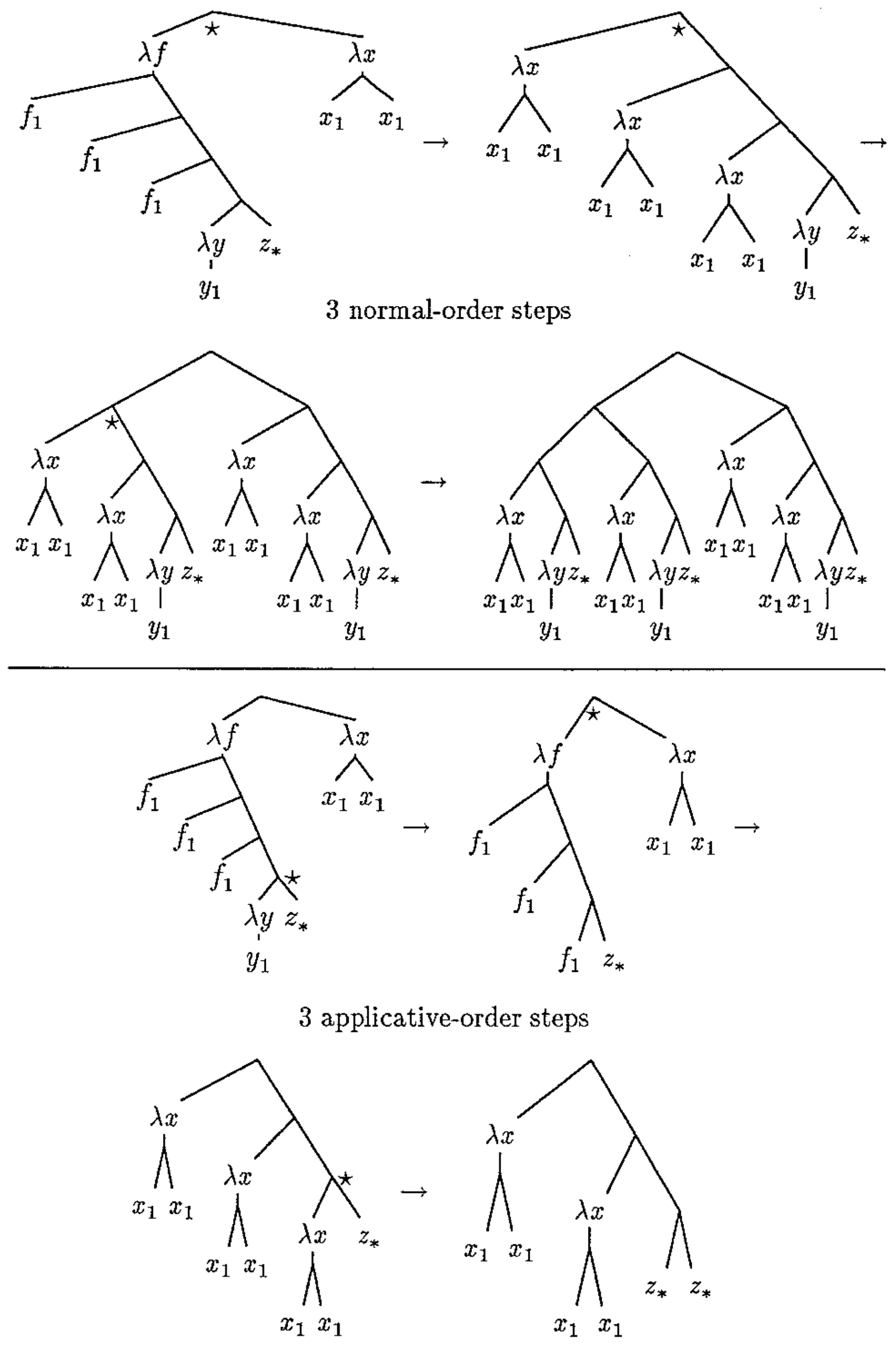


Figure 2.9: Function composition example

Chapter 3

Graph reduction: the λ_g-interpreter

This means that an argument is evaluated at most once, its evaluation being delayed until first needed. After Wadsworth this kind of 'lazy' evaluation has become a lifestyle.

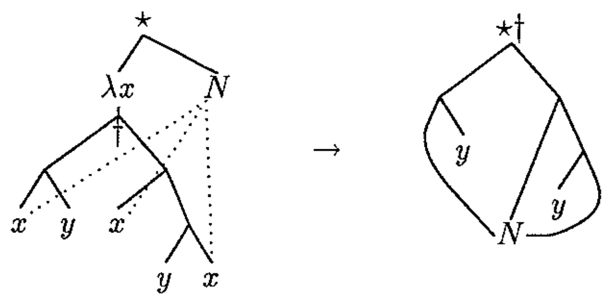
— Aiello and Prini (1981).

This dissertation studies the normal-order evaluation of the λ-calculus by comparing graph reduction with suspension-based reduction. The first part of this chapter presents a full graph-reduction interpreter that will serve for the graph-reduction half of the comparison. The next chapter presents the alternate interpreter.

Section 3.5 reviews parallel graph-reduction architectures. The main question there is: Why did the designers of those machines choose *graph* reduction? I dwell particularly on what they say about sharing.

For illustrative purposes, the interpreter in this chapter is encoded in ML. Appendix A provides a reader's guide to ML and describes some utility functions for graphs (Section A.2.2). The code reflects the non-functional, pointer-twiddling nature of graph reduction by using references (pointers), dereferencing, and assignments.

Introduction. This chapter introduces *graph reduction*, an implementation technique often used for functional languages, and presents a normal-



$$(\lambda x.\{((x\ y)\ (x\ (y\ x)))\}\ N) \quad\rightarrow\quad ((N\ y)\ (N\ (y\ N)))$$

Figure 3.1: Simple graph reduction

order graph reducer for the λ-calculus: a λ_g*-interpreter*. Wadsworth invented graph reduction for the pure λ-calculus [201];[1] its main virtue is that it provides the needed sharing for normal-order evaluation.

Wadsworth's fundamental insight was this: when faced with a substitution $M[x := N]$ during β-reduction, replace each instance of x in M with a *pointer* to N rather than a copy of N. Figure 3.1 shows a simple graph β-reduction, with dotted lines highlighting the intended substitutions. (The graphs do not use binding indices for reasons discussed in Section 3.4.1.)

Figure 3.1 illustrates two important things a graph reducer must do. The first is obvious: the rator's bound variables are replaced by pointers to the rand. The rand is *not* duplicated; the single copy is *shared* (space sharing).

The second thing is more subtle—the root of the redex (marked by $\star$ in Figure 3.1) is *overwritten* with the result of the reduction (the node marked $\dagger$). If the root node (node $\star$) has several pointers aimed at it, the overwriting lets them all "see" the result of the reduction—computation sharing. Graph reduction provides maximal sharing of both space and computation.

Top-level structure of a λ_g-interpreter. I now present the details of a λ_g-interpreter. A λ-calculus interpreter has three essential parts: a *search* strategy to find β-redexes, a procedure to *copy* shared rators (Section 3.3 describes this implementation concern), and β*-reduction* (substitution, mainly) to apply to the chosen redexes. Optionally, the interpreter may apply "*tidying*" transformations between reduction steps for efficiency reasons.

[1]Ironically, in the introduction of his thesis, Wadsworth said that he considered the work on semantics to be "more significant" [201, pages 2–3], yet he is certainly best known for inventing graph reduction.

```
(* Top-level loop: drives onestepG (λg-interpreter). Does the skipping over top-level
   λg-abstractions. Uses onestepG (page 36) and rm_indir_nodes (page 176).
*)
fun toplevG (ref(LamG(B,_,_,_))) = toplevG B
  | toplevG other                = real_toplevG other
and real_toplevG G =
let val done_in_G = onestepG G (* step forward *)
    val G'        = rm_indir_nodes G (* tidy things up *)
in if done_in_G then (* keep going *) real_toplevelG G' else G'
end
```

The normal-order search strategy of this λ_g-interpreter is a pre-order walk as would be used on the underlying λ-tree; however, subgraphs rooted at already-visited nodes are not revisited. The function onestepG (page 36) encodes this strategy; it is repeatedly invoked by toplevG (page 32) until onestepG indicates that no relevant redexes remain.

Copying of a shared rator before β-reduction is the job of lazy_copy (page 41) with substG (page 37) doing the subsequent substitution. In this λ_g-interpreter, the periodic removal of indirection nodes counts as "tidying," but I ignore this in comparisons of interpreters later on.

With that bird's-eye view in mind, I begin by describing the data structures that represent λ-terms in the λ_g-interpreter, then I present its constituent parts.

3.1 Graph structure and terminology

The basic data-structuring implication of graph reduction is that λ-terms are represented by (directed acyclic) *graphs*, not trees. (Cyclic graphs are sometimes used to represent recursive functions more efficiently.) When I describe graph-related things, I often add a g subscript; for example, "λ_g-graph," "λ_g-term" or "λ_g-interpreter."

A λ_g-graph is a directed graph consisting of a set of *nodes* and a set of directed *edges*. (In figures, direction on edges is from the higher node to the lower node unless an arrowhead shows otherwise.) λ_g-graph nodes represent the basic constructs of the λ-calculus in the obvious way; Figure 3.2 gives the ML definition of a λ_g-graph node; most of the fields are *updatable* (all the refs), because graph reduction modifies the graph in place. The following

```
(* Graph nodes for the λ_g-calculus; names commonly used are shown. *)
type Gnodeinfo =
    bool ref *          (* subbed; true if substituted in *)
    int ref *           (* refcnt; reference count [debugging only] *)
    bool ref *          (* visited; visited/marked? [housekeeping] *)
    (int * int) ref     (* x_y; x,y coords for anim [debugging only] *)
datatype Gnode
  = AppG of
        Gnode ref * (* M; rator *)
        Gnode ref * (* N; rand *)
        bool ref *  (* indir; true if a temporary indirection node *)
        Gnodeinfo   (* bits to keep around *)
  | LamG of
        Gnode ref * (* B; body *)
        int ref *   (* bndrID; binderID for backpointers *)
        string *    (* n; variable name: decorative *)
        Gnodeinfo
  | VarG of
        int ref *   (* bi; binderID: backpointer *)
        string *    (* n; variable name: decorative *)
        Gnodeinfo
```

Figure 3.2: Graph nodes' structure

kinds of nodes may exist:

AppG: Represents a λ-application; its left and right children are pointers to the rand and rator, respectively.

A subbed flag is set true when a node is the root of a graph that represents a substituted free expression; Section 3.3 discusses the reasons for this flag. A visited flag is set by the λ_g-interpreter when it wants to avoid re-visiting nodes. A reference count (i.e., number of pointers to the node) and a pair of (x, y) coordinates (used to make figures) are for debugging only.

An AppG may be temporarily turned into an *indirection node* by setting its indir flag. If set, the rator-pointer indicates the intended target.

LamG: Represents a λ-abstraction; it has a pointer to the λ-abstraction body. The LamG node has a unique integer *binder-ID*; variables can then match against this ID to see if they are bound to this λ-abstraction. (The support routines for these IDs are in Section A.2.2 (page 177).) The name on an abstraction is preserved (i.e., x for a λx), but it is purely decorative. The housekeeping fields are like those of AppGs.

VarG: Represents a variable; the important field is the binder-ID that identifies the LamG where the variable is bound (constants have an ID for which there is no matching LamG). The binder-ID is, in effect, a *backpointer* to the variable's binder. (See Section 3.4.1 for why binding indices cannot be used.) The housekeeping fields are like those of AppGs. The name is purely decorative, as usual.

Some terminology about λ_g-graphs is needed, especially for comparisons later on. A node has a *type*: three possible types are λ-applications, λ-abstractions, and variables bound to λ-abstractions. Nodes of these particular types are *plain* nodes; all nodes in a λ_g-graph are plain (making it a rather dull distinction at this point!).

If two plain nodes are directly connected by an edge, they are *g-connected*. All edges in a λ_g-graph are g-connections.

Definition 3.1 Two λ_g-graphs G_1 and G_2 are *λ_g-equivalent* if they are isomorphic graphs in which each corresponding pair of nodes $g_1 \in G_1$ and $g_2 \in G_2$ have the same type and the same subbed-flag value.

Converting between simple λ-terms and λ_g-graphs. A simple, tree-structured λ-term (like those in Chapter 2) *is* a λ_g-graph. (I ignore the ML type-conversion mechanics of replacing App, Lam, and Var nodes with AppG, LamG, and VarG nodes (respectively) and the messy conversion from binding indices to binder-ID backpointers as done in the code in Appendix A.2.2.)

The function graph2termT : Gnode ref $\rightarrow$ Term (page 67) converts a λ_g-graph to its linear-expansion λ-term. (In the ML code, I use the name "TermT" to indicate a simple λ-term made from λ-applications, λ-abstractions, and variables.) All the sharing in the λ_g-graph is unwound, producing a plain λ-term with no sharing.

3.2 Finding a redex and β_g-reduction

As already suggested, walking a λ_g-graph to find the next redex (in pre-order) is fundamentally the same as walking a λ-tree, except that previously-visited subgraphs need not be re-walked. The ML function onestepG : Gnode ref $\rightarrow$ bool (page 36) finds the next redex and then modifies the λ-graph appropriately, returning a boolean indicating whether a reduction took place. (I defer the problem of shared rators to Section 3.3.)

The important lines in onestepG are the calls to lazy_copy (described in the next section) and substG (page 37) that does a substitution: all variables with a binder-ID matching that of the rator λ_g-abstraction are replaced with a *pointer* to the rand N. SubstG also reports the number of substitutions done.

Purely for reasons of compatibility with the interpreter in the next chapter, the λ_g-interpreter handles trivial reductions specially. If the rand is a single variable, i.e., the redex is $(\lambda x.\{B\}\ y)$, the substitution of y is actually done and the substitutions' subbed flag are *not* set. Sharing a variable is trivial and serves no purpose. This case is handled by substG (page 37).

Similarly, if the rator-body is a single bound variable, i.e., the redex is $(\lambda x.\{x\}\ P)$, then P is the result (as usual) but its subbed flag is *left alone*. Section 4.5.2 explains about why these special cases facilitate exact comparisons of interpreters.

OnestepS achieves the effect of overwriting the root of the redex by turning it into an indirection node, a common technique. It is simpler than looking through the whole λ_g-graph to find pointers to the redex and re-aiming them. I will generally ignore indirection nodes; toplevG (page 32) removes them between steps for simplicity's sake.

```
(* onestepG : Gnode ref → bool  is passed a pointer to (part of) a graph; it finds the
   first redex in it and reduces it (modifying the λg-graph in place). It reports whether
   or not a reduction was done; it reduces to WBNF.
   It uses incr_refcnt (page 175), set_subbed (page 175), lazy_copy (page 41), and substG
   (page 37).
*)
fun onestepG (ref (AppG(M,N,(ref true),_))) = raise unexpected_indirection_node
  | onestepG (ref (AppG(M as ref (LamG(B as ref (VarG(vbi,n,(_,_,_,_))),
                                        lbi,_,(_,lrefcnt,_,_))),
                  N,indir,(subbed,_,_,_)))) =
    (* beta redex: special case of a trivial rator body *)
    ( indir := true;           (* this AppG now an indirection node! *)
      set_subbed false N;      (* the reason for the special case *)
      if ((!vbi) = (!lbi)) then (* bound *)
          (M := (!N))
      else                     (* useless reduction *)
          (M := (!B));
      true
    )
  | onestepG (G as ref (AppG(M as ref (LamG(B,si,_,(_,lrefcnt,_,_))),
                              N,indir,(subbed,_,_,_)))) =
    (* beta redex; special case of a trivial rand handled in substG *)
    let
        val _                  = incr_refcnt ~1 M; (* will lose AppG refs *)
        val _                  = incr_refcnt ~1 N;
        val _                  = set_subbed true N; (* the key to laziness! *)
        val Blzcpy             = lazy_copy B;
        val (instance, no_subs) = (substG (!si) N Blzcpy);
    in (
        indir := true; (* this AppG now an indirection node! *)
        incr_refcnt ~1 M; (* whatever M is will have one less ref *)
        (M := (!instance)); (* redex overwritten! *)
        true
    ) end
  | onestepG (ref (AppG(M,N,_,_)))  = (* not a redex *)
      (onestepG M) orelse (onestepG N)
 (* onestepG (ref (LamG(B,_,_,_))) = onestepG B, if BNF were possible... *)
  | onestepG other                  = false (* a LamG or a VarG *)
```

```
(* substG : int → Gnode ref → Gnode ref → Gnode ref * int  fills in bound variables
   (those backpointing to bndrID) with pointers to sub_with, or—if sub_with is just a
   variable—with a copy of the variable itself (trivial substitution). substG also returns
   the number of substitutions done.
   SubstG takes the usual precautions against revisiting subgraphs (with mk_graph_visited
   (page 175)); not visiting subbed nodes would work as well. subG is the local function
   that goes on to do all the work.
*)
and substG bndrID sub_with node =
let (* cases with 'subbed' and 'visited' false given first *)
   fun subG bndrID sub_with (G as ref (AppG(M,N,(ref true),_))) =
       (perr("unexpected indir node:"^unparse(graph2termT G));
        subG bndrID sub_with M)
     | subG bndrID sub_w (G as ref (AppG(M,N,_,(_,_,visited as (ref false),_)))) =
       let val _         = (visited := true)
           val (M',mc) = (subG bndrID sub_w M)
           val (N',nc)  = (subG bndrID sub_w N)
       in (G, (mc:int)+(nc:int)) end
     | subG bndrID sub_with (G as ref (LamG(B,_,_,(_,_,visited as (ref false),_)))) =
       (* a copy of a LamG node needs all-new binderIDs *)
       let val _ = (visited := true)
           val (B',bc) = (subG bndrID sub_with B)
       in (G, bc) end
     | subG bndrID sub_with (G as ref (VarG(si,_,(_,_,visited as (ref false),_)))) =
       ((visited := true);
        if (!si = bndrID) then ( (* substituting! *)
           case sub_with (* but not if trivial... *)
              of (ref (VarG(ssi,sn,(_,_,_,_)))) ⇒ ( (* subbed is true *)
                   ((ref (VarG(ref (!ssi),sn,(ref true,ref 1,ref false,ref (0,0))))),
                    1)
                 )
               | _ ⇒ ( (* non-trivial substitution *)
                   incr_refcnt 1 sub_with;
                   G := !sub_with;
                   (G, 1))
        ) else (* just copying *)
           (G, 0))
    (* finally, if 'visited' is true... *)
     | subG bndrID sub_with already_visited = (already_visited, 0)
in (mk_graph_visited false node; subG bndrID sub_with node) end
```

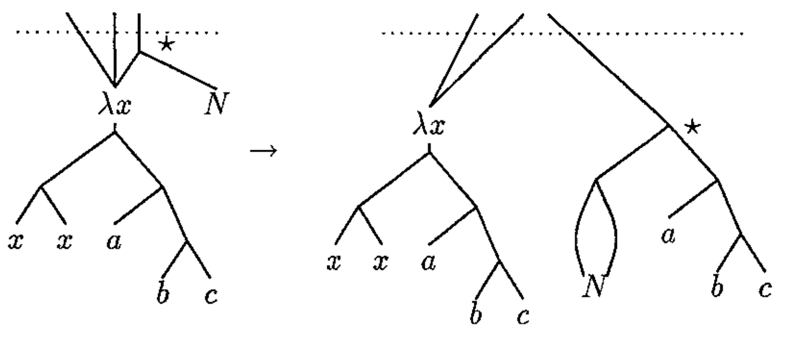


Figure 3.3: Graph reduction with copying

3.3 Sharing free expressions and lazy copying

There is more to graph reduction than substitution with pointers to, rather than copies of, a λ_g-term. Some copying is *unavoidable*. The problematic case arises when the rator of a redex (a λ-abstraction) is shared. An abstraction $\lambda x.M$ is a "template" for a reduction; M defines the "shape" or "structure" of the result, with the bound variables of λx being placeholders to indicate where the rand should be "plugged in." To fill in the placeholders of a shared rator is to use up the other sharers' template. Consequently, shared rators *require* some copying. Figure 3.3 shows a reduction, marked with a $\star$, in which the rator is shared (with three pointers to it). A complete copy of the rator is kept for possible later use.

Wadsworth sought to maximize sharing of space and computation. For those criteria, the reduction in Figure 3.3 copies too much. In particular, the sub-term $(a\ (b\ c))$ includes no bound variable x, so one copy may be shared among all instances of the λ-abstraction's body. $(a\ (b\ c))$ is a *maximal free expression* (MFE) in the term $\lambda x.\{((x\ x)\ (a\ (b\ c)))\}$. (Other free expressions include c and $(b\ c)$, but neither is maximal.) Figure 3.4 shows the same reduction as Figure 3.3, but with the MFE $(a\ (b\ c))$ shared.

Detecting and sharing MFEs gives the most possible sharing, and therefore the least copying, but it is an expensive task for run-time.[2] Arvind et al. [9] examined the alternatives in some detail; they say an interpreter like Wadsworth's that detects and uses MFEs does *fully lazy* copying.

Arvind et al. also discuss *lazy-copying* interpreters, citing Henderson

[2]Also, the "most possible" sharing is not necessarily the "best possible;" Peyton Jones's book on sequential implementations of graph reduction has a section on "Excessive Sharing" [165, Section 23.4.2]!

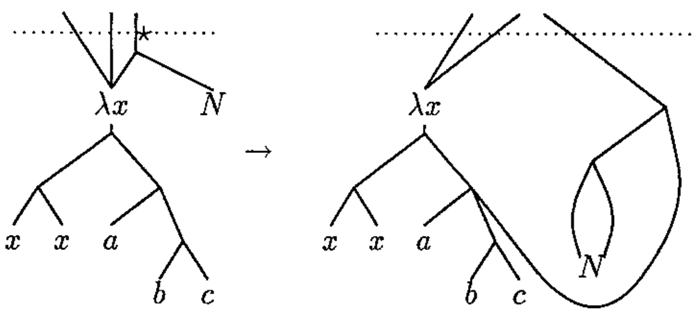


Figure 3.4: Graph reduction with a shared MFE

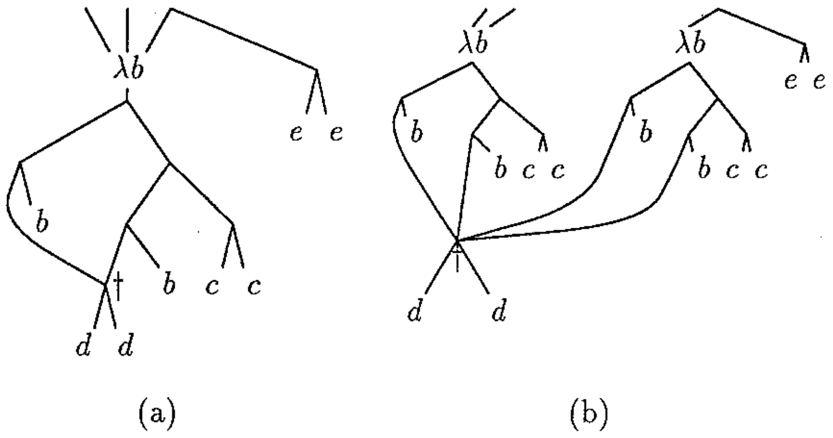


Figure 3.5: A lazy copy

and Morris's evaluator as an example [90]. Lazy copying does not detect MFEs; instead, it exploits a simple observation: If the term $(\lambda a.\{\lambda b.\{M\}\}\ N)$ is reduced to $\lambda b.\{M[a := N]\}$, then N is free in $\lambda b.\{M[a := N]\}$, because variables in N could not "see" the λb abstraction to be bound to it. I call it a *substituted free expression* (SFE). Lazy-copying flags SFEs when first encountered and avoids copying them later on. Put colloquially, lazy copying does not seek out free expressions, but it makes good use of the ones that come its way. In figures, daggers † mark the roots of the SFEs; in the ML code, `subbed` flags are set to `true`.

Lazy copying leads to less sharing than fully-lazy copying. Figure 3.5 shows (a) a term with a redex at the top and (b) the term after a lazy-copy of the shared rator (the β-reduction itself would follow). Laziness manifests itself in the sharing of be-daggered $(d\ d)$; fully-lazy copying would detect the free expression $(c\ c)$ and avoid copying it also. The function lazy_copy : Gnode ref $\rightarrow$ Gnode ref (page 41) is an implementation.

Arvind et al.'s main result that is relevant here is that lazy copying gives the same sharing as fully-lazy copying if the initial λ-terms have their free expressions λ-lifted, as in conversion to supercombinators, for example [105]. (Section 6.5 happens to give an example of such a conversion.) With this assurance in mind, I consider lazy or fully-lazy copying to be equally acceptable. The interpreters in this chapter and the next do (non-fully) lazy copying of shared rators.

That is the good news. The bad news is that lazying copying can give incorrect results on λ_g-graphs with redexes inside (non-top-level) λ_g-abstractions. Put another way, it is correct only for reduction to weak β-normal form (WBNF). Figure 3.6a shows a λ_g-graph after one reduction († on the SFE). The second reduction is inside the λz abstraction, as Figure 3.6b shows. The third redex will be the one marked with a $\star$; because its rator is shared, a lazy copy must precede the reduction. Because $(z\ z)$ is a be-daggered alleged SFE, it will *not* be copied. But the z's mean that $(z\ z)$ is *not* free in the λz abstraction, and they must not be shared.

The λ_g-interpreter on the recurring example. Figure 3.7 shows all the steps of the graph reduction of the recurring example; the daggers indicate SFEs.

3.4 More on variable bindings

3.4.1 Graph reduction with binding indices

Graph reduction is an implementation technique for the λ-calculus that has desirable sharing properties; binding indices are an effective solution to the problems of variable naming: can the two techniques be used together? Oddly enough, for normal-order evaluation the answer is "No."

Figure 3.8 shows a reduction done two ways, by (a) simple tree reduction and by (b) graph reduction. There are two substitutions for x; for tree reduction, the binding indices in the substituted terms *differ*. In particular, the free variables y and z in the two copies of rand are now different distances from their binders. The problem is that λ-terms are represented by graphs, so binding paths are no longer unique.

If the rand is to be shared, then something must be done to balance the

```
(* lazy_copy : Gnode ref → Gnode ref  copies the subgraph of connected unsubbed nodes
   rooted at its argument node. That is, when a subbed node representing an SFE is
   seen, a pointer to that node is returned and the copy does not proceed inside that
   subgraph.
   The function chg_bndrIDs (page 177) is used to fix backpointer binderIDs in the copied
   parts of graphs.
*)
and lazy_copy node =
let (* cases with 'subbed' and 'visited' false given first *)
  fun lzcp (G as ref (AppG(M,N,(ref true),_))) =
       (perr("unexpected indir node:"^unparse(graph2termT G)); lzcp M)
    | lzcp (G as ref (AppG(M,N,_,((ref false),_,visited as (ref false),x_y)))) =
       let val _         = (visited := true) (* just copying an AppG node *)
           val (M', N') = (lzcp M, lzcp N)
       in (ref (AppG(M', N', ref false, (ref false, ref 1, ref false, ref (!x_y))))) end
    | lzcp (G as ref (LamG(B,old_bndrID,n,((ref false),_,visited as (ref false),_)))) =
       (* a copy of a LamG node needs all-new binderIDs *)
       let val _              = (visited := true)
           val new_bndrID = next_ID ()
           val B'             = (lzcp B)
           val _              = chg_bndrIDs (!old_bndrID) new_bndrID B'
       in (ref (LamG(B', ref new_bndrID, n, (ref false,ref 1,ref false,ref (0,0)))))
       end
    | lzcp (G as ref (VarG(si,n,((ref false),_,visited as (ref false),x_y)))) =
       (visited := true;
        (ref (VarG(ref (!si), n, (ref false, ref 1, ref false, ref (!x_y))))))
   (* now the cases with 'subbed' true but 'visited' false *)
    | lzcp (G as ref (AppG(_,_,_,((ref true),_,visited as (ref false),_)))) =
       let val new_ptr = (ref (VarG(ref 0,"",(ref false,ref 1,ref false,ref (0,0)))))
       in ( new_ptr := !G; new_ptr) end
    | lzcp (G as ref (LamG(_,_,_,((ref true),_,visited as (ref false),_)))) =
       let val new_ptr = (ref (VarG(ref 0,"",(ref false,ref 1,ref false,ref (0,0)))))
       in ( new_ptr := !G; new_ptr) end
    | lzcp (G as ref (VarG(_,_,((ref true),_,visited as (ref false),_)))) =
       let val new_ptr = (ref (VarG(ref 0,"",(ref false,ref 1,ref false,ref (0,0)))))
       in ( new_ptr := !G; new_ptr) end
   (* finally, if 'visited' is true... *)
    | lzcp already_visited =
       (perr("already visited:"^unparse(graph2termT already_visited));
        raise visited_when_copying)
in (mk_graph_visited false node; lzcp node) end
```

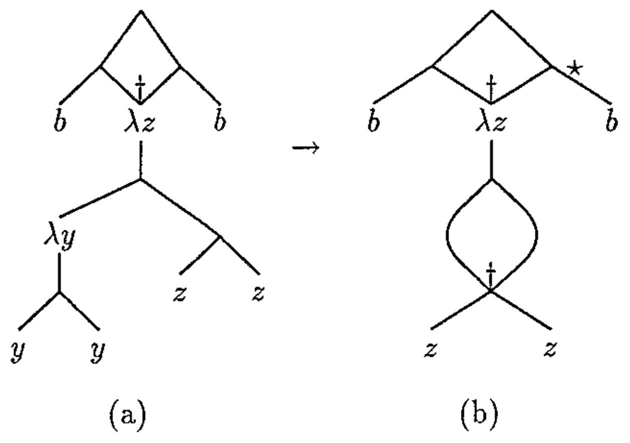


Figure 3.6: Lazying copying will not work for non-weak reduction

binding-path lengths for the would-be substitutions. One can imagine inserting path-balancing nodes into the graph; however, allowing these nodes to appear anywhere in the graph complicates the basic definitions of the λ-calculus (e.g., β-reduction). Berkling's variant of binding indices [27; 31; 32] uses such "unbinding" operators, but they do not help with graph reduction. The description of a "k$\lambda\pi$ calculus" in Schlütter's dissertation is a clear, extended explanation of a system that includes unbinding [180, pages 87–119].

Graph reduction and binding indices can be used together if no reduction is done inside λ-abstractions, i.e., to a weaker normal form. Peyton Jones's book on implementation discusses the relevant techniques, especially for weak head-normal form (WHNF) [165, pages 198–199].

The difficulties with binding indices for normal-order evaluation are the reason I use Wadsworth's backpointers in the λ_g-interpreter, implemented with global integer binder-IDs (Section A.2.2) [201]. It is esthetically unpleasing not to have a binding-indices graph-reducer to compare with my binding-indices tree-reducer of the next chapter, but the discrepancy is not important.

3.4.2 Wadsworth's use of backpointers

Wadsworth suggests a clever use of backpointers and indirection nodes that not only avoids variable-naming problems but also avoids having to search for bound variables during β_g-reduction [201, pages 176–180]; Figure 3.9 gives an example, with backpointers shown by dashed arcs.

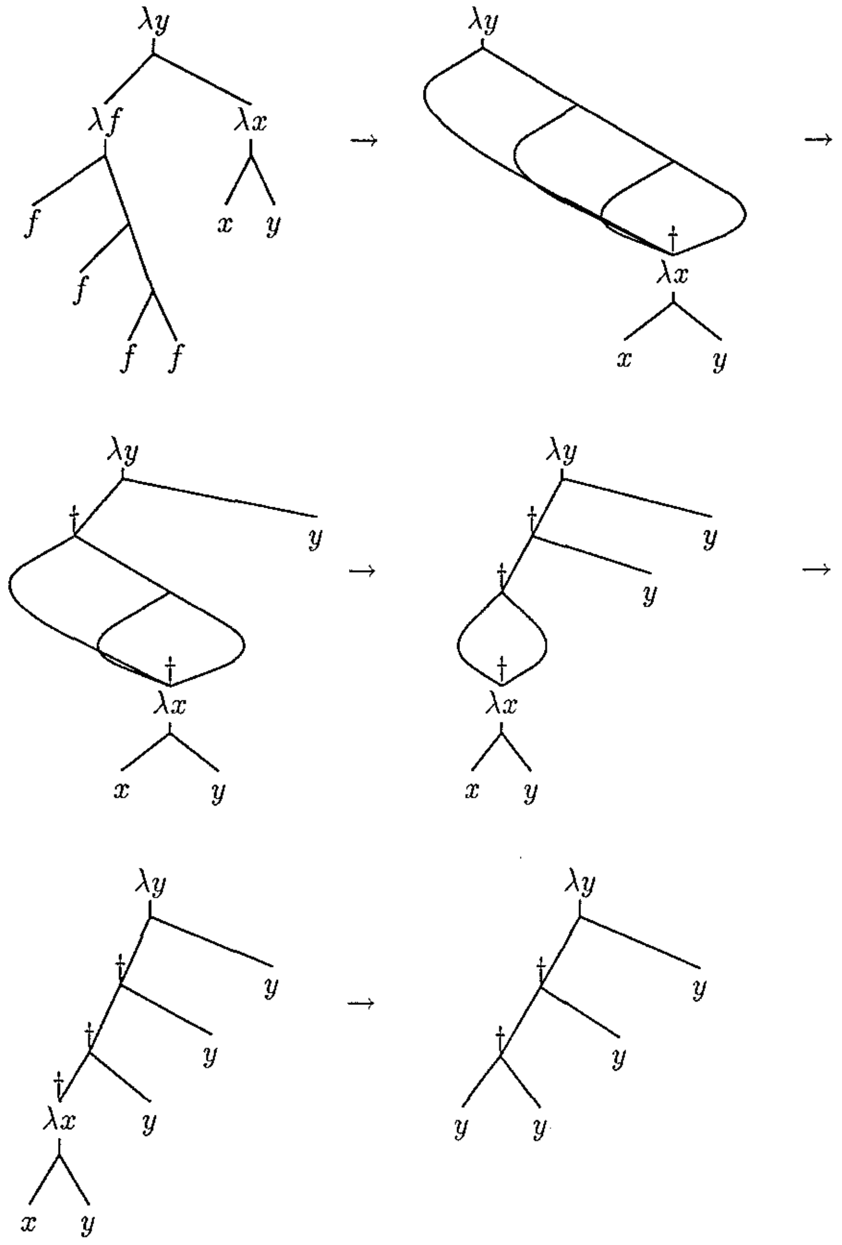
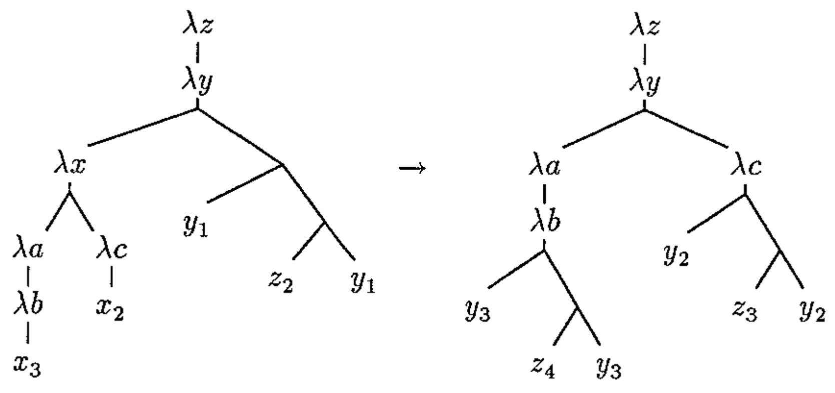
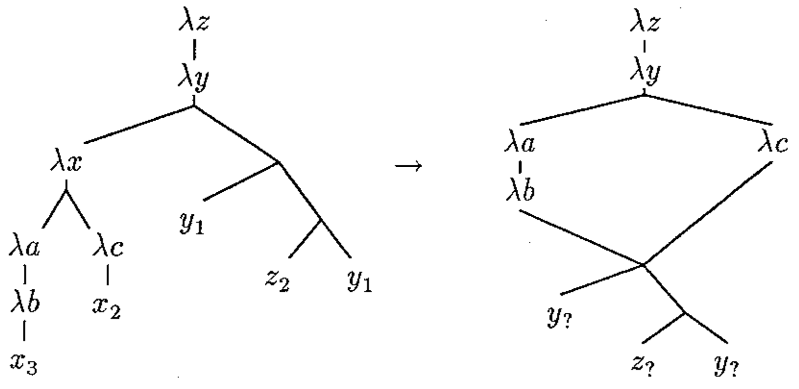


Figure 3.7: The λ_g-interpreter on the recurring example



(a) by tree reduction



(b) by graph reduction?

Figure 3.8: Graph reduction with the name-free λ-calculus

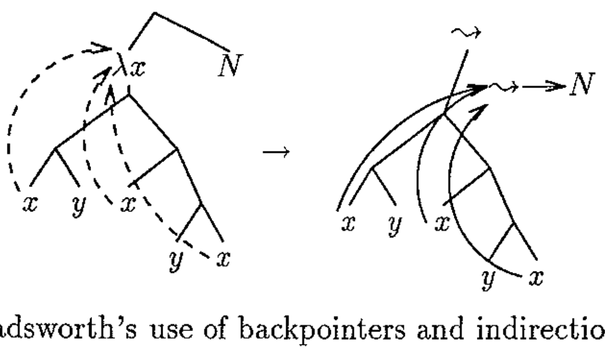


Figure 3.9: Wadsworth's use of backpointers and indirection nodes ($\rightsquigarrow$)

As usual, the root of the redex becomes an indirection node (shown by $\rightsquigarrow$). The novelties are that the rator's λ-node *also* becomes an indirection node and that the backpointers are treated as "real" pointers thereafter. For this to work, the target of a backpointer must be examined before one knows whether it should be followed or not. It may not be very efficient, but it is a neat idea; Berkling provides a critique in his 1986 paper [25, page 29].

3.5 Graph-reduction architectures and sharing

This section sketches some of the parallel architectures that have been designed to support graph reduction and reviews some of the designers' comments on their experience. I want to know *why* they chose graph reduction as their basic computational model.

The ALICE parallel graph-reduction machine (Imperial College, London) [56; 88; 49] and its follow-on, Flagship (University of Manchester and ICL also collaborating) [205], are an instructive pair of designs. ALICE was first reported in 1981; a prototype was running in the summer of 1986; its designers' practical experience merits close attention.

In ALICE, each node of a program graph is represented by a *packet* with a globally-known address; the packet includes the node type, the addresses of the nodes to which this node is connected, and other information (status bits, etc.). All the packets representing the program graph are in the *packet pool* (distributed) memory, shown by 'M' units in Figure 3.10 (the numbers of units shown do not reflect any real configuration).[3] Across the switching network sit several *packet processors* ('P' units). Each processor repeatedly fishes an active packet out of the pool and tries to do the rewrite suggested by the packet type. Further packets may have to be read, and, eventually, a set of packets representing the result will be thrown back into the pool. Because some packet types require evaluated arguments (e.g., arithmetic primitives), there is a mechanism to suspend active rewrites and to re-activate them when the required preliminaries have been done.

Flagship is a successor to ALICE that also draws heavily on the experience of the Manchester dataflow machine group (who also built a prototype). The basic architecture chosen for Flagship, shown in Figure 3.11,[4] reflects the

[3]Keller et al. call this a "dancehall" system organization, with "processors lined up along one side of a large dancehall, and memories along the other, with a network of switches in between" [114, page 411].

[4]This architecture is much closer to a "boudoir" organization, in which "each processor

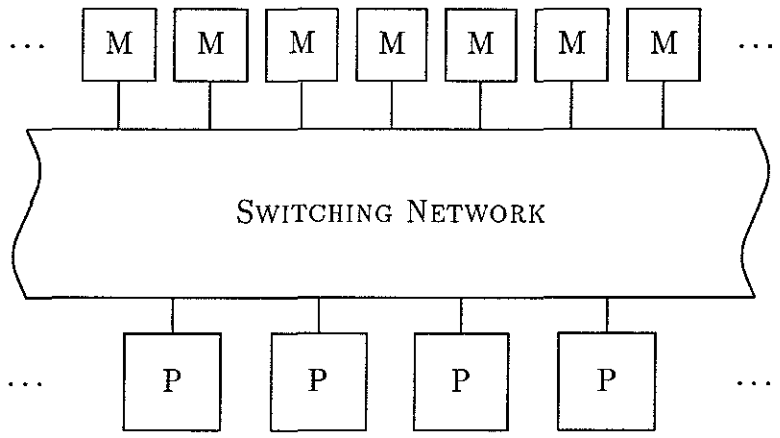


Figure 3.10: ALICE machine organization ("dancehall")

designers' foremost criticism of its predecessors: Having a switching network between the processors and memories requires that the network have very high bandwidth and that programs have massive parallelism to overcome long network latencies. Flagship designers have great concern for *locality*; they aim for "90% locality"—"9 out of 10 store accesses made by a rewrite should be to the local processor." [205, page 127]. The machine uses local caching of non-local parts of the program graph, so that rewriting itself is entirely local. Still, the designers concede that preserving locality is a "difficult problem, which forms a major aspect of Flagship research" [205, page 128], and they advocate that algorithm, language, and compiler design, plus dynamic mechanisms, be examined specifically to *enhance locality*. The approach taken to support lazy functional languages is supercombinators evaluated to WHNF; this means they look for large self-contained rewrites whose graph manipulations can be encoded as big swatches of von-Neumann code suitable for local execution [208]. (I should mention that the Flagship group found a supercombinator approach to be "ten to a hundred times more efficient" than a straight λ-calculus graph reducer [208; 206]; Berkling disagrees [25; 30].) Flagship takes locality seriously indeed: they want to preserve the "fine-grain" computational model of graph reduction but to avoid the considerable work a literal implementation of that model must entail [202].

There is no evidence that ALICE's designers considered a computational

is closely paired with a memory" [114].

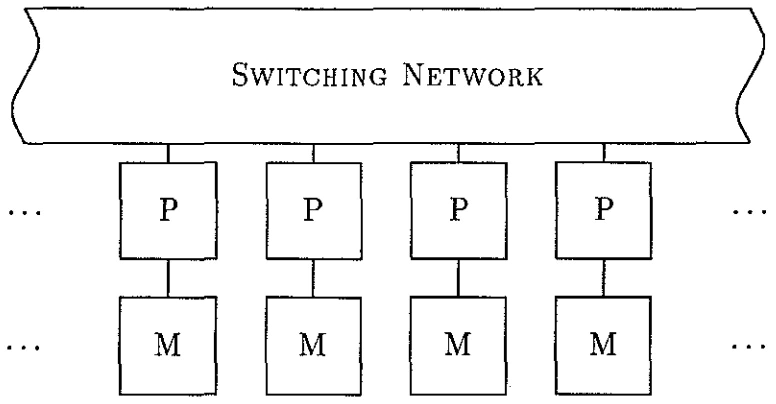


Figure 3.11: Flagship machine organization ("boudoir")

model other than graph reduction, which is understandable given their unswerving commitment to lazy functional languages. By the time of Flagship, its designers were interested in supporting "graph rewriting," a more general mechanism discussed in Section 3.6. They note that graph reduction subsumes string reduction and "that string reduction can be readily implemented in any graph reduction machine structure in those circumstances where it may be worthwhile" [203, page 9].

Though they favor graph reduction as a computational model, the Flagship designers are plain-spoken about its costs. They state, unequivocally, "Graph reduction has a fundamental requirement for a global memory space at the level of an abstract machine architecture" [204, page 266]. Put another way, the "notion of node identity is an essential feature of graph [reduction] (as distinct from [tree reduction])" [207, section 2]. For a parallel design, "it is inevitable that much of the physical memory will not be immediately accessible by any single processor. The performance of any parallel machine is critically dependent on the way in which remote access is handled" [204, page 266].

The work of Keller et al. in Utah, first reported in 1979, shows some of the same trends as ALICE/Flagship. AMPS was the first parallel graph-reduction machine from the Utah group; it was oriented to symbolic computation and intended to run LISP [115]. AMPS had a "boudoir"-ish system organization (Figure 3.11), with a tree-structured switching network; the Processor/Memory units are connected to the leaves of the network, implementing a virtual global address space. The tree-node hardware supports packet-switched communication, as well as load-balancing (hardware-assisted load-balancing is a hallmark of the Utah designs). The AMPS designers, though influenced by fine-grained dataflow approaches, sought "task-level" parallelism and used caching, saying "One of the most important concepts of our architecture is an attempt to improve performance by exploiting locality of information flow" [115, page 614].

AMPS supported graph reduction to allow programming with infinite structures and also because "by exploiting the richer connectivity of graphs, we can avoid much of the combinatorial explosion which takes place in purely string-oriented reduction machines" [115, page 616]. The mechanism of rewriting graphs is not fundamentally different from ALICE or Flagship: successive rewritings of "code words" represent graph-node manipulations.

Rediflow, "a collection of ideas relating to multiprocessor system design and attendant software capabilities," reflects the development of the Utah group's thinking [114]. They moved toward a "hybrid model" that supports "reduction, dataflow, and von Neumann processes." The physical configuration changed from AMPS's tree to a rectangular mesh of "Xputers," but the boudoiresque, "medium grain" direct implementation of graph reduction is unchanged. The signature concern for load distribution and balancing is more evident [136].

Rediflow II is a "proposed multiprocessor architecture based on graph reduction" [116]. The most notable change is away from a direct "more-interpretive, graph reduction model" to a "PSCED"[5] approach, "an efficient method for the integration of sequential code" [116, page 204] (the authors say the PSCED idea is similar to SECD-m [1] or the Multilisp implementation [85; 86]). As with supercombinator machines, the goal is to have a successful non-von Neumann machine by shoehorning as much von-Neumann-style execution into the implementation as possible.

Paul Hudak (Ph.D. from Utah) and his cohorts had a similar goal in their work on "serial combinators" [102; 103]: to compile functional programs into combinators for which the internal workings are entirely sequential, but without a loss of parallelism in the program. The serial code is gathered into bundles that can be executed cost-effectively by a single processor.

Another British parallel graph-reduction machine is GRIP, under the direction of Simon Peyton Jones [169; 168]. GRIP uses a bus for its communication medium (trading scalability for modest expense and low latency) and hopes to be a cost-effective dozens-of-processors computer. For lazy

[5]A sort-of acronym for "parallel SECD machine," pronounced "sked."

functional languages, GRIP intends to use supercombinators to evaluate to WHNF, based on a parallel extension to the G-machine model (building on the G-machine compiler technology for sequential computers [165]). Precious bus bandwidth is conserved by using "intelligent memory units" that do higher-level operations on the graph data they hold (e.g., create variable-sized nodes and do garbage collection). GRIP also has a central system-management processor.

Peyton Jones also draws attention to the importance of locality, saying, "One issue dominates all others: *how can a high degree of spatial locality be achieved simultaneously with a high degree of processor utilisation?*" (his emphasis) [167, page 182]. He goes on to say that a "bus was chosen specifically to make the locality issue less pressing" [167, page 183].

The Dutch Parallel Reduction Machine project is another significant broad-based effort to support functional programming [23; 91]. The designers use graphs for sharing reasons, saying "graphs are an essential part in any implementation" [23, page 264]. They aim for "coarse-grain" reduction (at the level of "jobs"), letting a standard fast sequential reducer do the "fine-grain" reductions; they use annotations to help decompose programs to jobs. Most interestingly, they give up a global address space and do not share at the job level [200]. Instead, they use a "sandwich strategy" of reduction that evaluates shared jobs first and then *copies* the results to the sharing reducers' local address spaces. Thus, there is "a kind of string-reduction" [200, page 301] at the job level, with conventional graph reduction at lower levels.

Another design that uses distributed memory to implement a global address space is the HDG-Machine at the GEC Research Centre in England [33; 40; 41]; it is implemented on a network of transputers with distributed memories [210]. The underlying model of the HDG-Machine is the use of *evaluation transformers*, in which each function argument is analyzed at compile-time to decide how far it will need to be reduced. This information then guides the spawning and management of tasks in the machine. Terms are shared until reduced to WHNF; after that, they may be copied freely. Fast context-switching is essential in this machine, so stack frames are kept in the heap, and a context-switch is a simple pointer manipulation.

Several other proposed machines for graph reduction restrict themselves to combinator reduction. Combinators are pure rewrite rules, unencumbered by free/bound-variable difficulties, and mechanisms can be constructed that give the *effect* of going through all the λ-calculus reductions, but avoiding the intermediate steps. With super- or serial-combinators, the goal is to synthesize program-specific large combinators; again, to avoid intermediate steps. This approach is well-represented by already-mentioned machines. With fixed sets of combinators (e.g., S, K, I), the approach is to build them directly into the hardware. Sequential combinator-reducing machines have included SKIM [47], SKIM II [196], the CURRY chip [173], and NORMA [179]. TIGRE is a threaded combinator-reducing interpreter in which the graph is built partly from *instructions*, so that the graph itself is actually executed [127]. Parallel combinator machines include COBWEB [87; 34] and COBWEB-2 (which uses combinator-equivalent "director strings") [6; 7].

The MaRS project in France, with its background in aeronautics and dataflow architectures, proposed a combinator-based parallel graph-reduction machine [44; 48]. Of their graph-reduction choice, the designers say,

> [O]ne can distinguish two kinds of reductions, string reduction where expressions are formed of literals and values, and graph reduction, where expressions are constituted of literals and references. In our opinion, graph reduction is more interesting for the following reasons: because of the notion of reference, graph reduction allows the sharing of computation, but it also provides an efficient way for manipulating large data structures. Graphs are uniform representations of suspensions (closures) as well as of shared data structures [44, page 162].

There are still more proposals for "parallel graph reduction" machines; however, they do not speak out about the basic graph-reduction design choice. Amamiya proposes a heavily dataflow-influenced design that tries to use eager evaluation when possible [5]. (Generally speaking, reduction-machine designers have a great debt to their dataflow-machine predecessors and colleagues.) The G-machine group has designed a parallel variant, the $<\nu$,G$>$-machine [11]. Numao and Shimura give a graph manipulation scheme for a system of disjoint graph reducers not sharing a common address space but communicating by CSP-style message passing [156]. Traub proposed an "abstract" parallel graph reduction machine, to help separate and clarify the *many* design issues in such architectures [197].

There are other major issues associated with graph reduction besides sharing, notably garbage collection and load distribution. These questions are beyond the scope of this dissertation, but I will again quote the experienced Manchester group on the cost of the global address space that graph reduction requires: "Graph Reduction requires continuous storage allocation and reclamation. It is inevitable that our distributed global address space will make these tasks more difficult" [204, page 271].

It may well be true that the use of graph reduction has been an item of heated debate within the research groups that have chosen it; if so, the debate is not reflected in the literature. My speculation is that graph reduction is used because it is the best-known model that supports lazy functional programming; threatened exponential blow-ups of "string reduction" have kept researchers well away from that set of choices. Also, graph reduction offers plenty of sharing; however, designers do not strive for maximal sharing, suggesting it is not a critical design criterion. Most designers point out the importance of locality, with the experienced groups being all the more emphatic. Exotic and clever solutions like the Dutch "sandwich strategy" are used to overcome the inherent indifference to locality that is implicit to graph reduction. Given the modest success of parallel graph-reduction machines so far, I would suggest that some re-thinking is in order.

Some close cousins of graph reduction and architectures based on it use "closure-based reduction" or related schemes. I review that work in Section 4.8.3.

3.6 Graph rewriting

I have concentrated strictly on graph reduction as a technique for implementing lazy functional languages, but I should mention that many "graph reduction" researchers, including several cited above, take a broader view—in this context, it is usually called *graph rewriting*.

Graph rewriting sits on the well-explored theoretical base of *term rewriting systems*, which are sets of rewrite rules on terms (e.g., S, K, I combinators can be described as a TRS). Klop [123] and Dershowitz [60] provide good introductions to TRSs. Drawing on the field of graph grammars, graph rewriting considers sets of rewrite rules that operate on graphs. Because graph rewriting can represent a broad range of computational models (including both functional and logic programming), as well as many machine-level graph-twiddling tricks that a compiler might do, and because graph rewriting is amenable to formal treatment, it is suitable as an intermediate form and compiler target language for parallel computers.

The two main groups defining such languages for parallel machines are the Dactl group, at Manchester and East Anglia [74; 75; 77; 78; 76; 73; 119; 120], and the Lean group, part of the Dutch Parallel Reduction Machine project [20; 21; 22; 38]. FLIC is Peyton Jones's entry in the intermediate language field [166].

Graph rewriting is significant, as it helps to broaden our understanding of fundamental issues in graph-based computation; however, it has not affected my work, because I am not making claims about anything but the normal-order evaluation of the pure λ-calculus.

Chapter 4

Reduction with suspensions: the λ_s-interpreter

Suspension and reduction are ... remedies against the iniquitous or ill-founded decrees of inferior judges ...
— John Erskine, *An Institute of the Law of Scotland* (1773).

The previous chapter described the λ_g-interpreter, for the normal-order graph reduction of the λ-calculus. This chapter sets forth an alternate λ_s*-interpreter* that does the same reductions step-for-step but keeps λ-terms in *tree* form. Using a tree representation could provide better opportunity to preserve locality in a parallel implementation. To illustrate this benefit, Chapter 5 explains how an FFP Machine (FFPM) implementation would work.

Sections 4.1–4.5 present the λ_s-interpreter; Section 4.7 brings together the results of the previous sections to prove the λ_s-interpreter's equivalence to the graph-reduction λ_g-interpreter; since the λ_g-interpreter is known to be a correct implementation of the λ-calculus, the λ_s-interpreter must be as well. Section 4.8 examines the relationship of this work to others' efforts.

I have written a working λ_s-interpreter in a functional style, using ML. Most of the code is included in this chapter for illustrative purposes; it uses some utility functions described in Section A.2.1.

Introduction. This section describes an interpreter for the λ-calculus that manipulates ordinary λ-terms augmented with an additional element, the *suspension*—hence the name λ_s*-interpreter* (and the s subscript).

Wadsworth showed that graph reduction is a correct implementation of the normal-order evaluation of the pure λ-calculus [201]. If I can prove that the λ_s-interpreter does the "same thing" as the graph-reducing λ_g-interpreter on each reduction step, then it follows that the λ_s-interpreter is also a correct evaluator for the λ-calculus. Moreover, if the two interpreters work in lockstep, then it is relatively easy to compare their time and space complexities as well. This chapter is concerned with the λ_s-interpreter itself and its correctness. I defer complexity questions until Section 5.3, after the various operations' costs have been determined.

General battle plan. The diagram in Figure 4.1 suggests my broad claim, most of which I do *not* pursue: given a plain (non-graph) λ-term T_t^0 that (presumably) reduces to its weak β-normal form (WBNF) T_t^n in n *graph-reduction* steps, then the following do also:

- the conventional interpreter, eval_WBNF (page 25) (which will do at least n reduction steps);
- at least n calls to the plain-tree-reducing onestepT (page 23) (the asterisk superscripts in Figure 4.1 denote "zero or more" calls);
- n graph-reduction steps, as in the λ_g-interpreter of Chapter 3 (onestepG (page 36) is an encoding of the main routine); or
- n suspension-ridden steps of the λ_s-interpreter of Chapter 4 (onestepS (page 72) is an encoding of the main routine).

The initial terms T_g^0, T_t^0, and T_s^0 are trivially related.

I use a "super-toplevel" ML procedure that does what Figure 4.1 suggests: it runs all the interpreters together in lockstep, converting and comparing terms along the way, using the functions term2this and that2term of Figure 4.1 to convert between representations. The "super-toplevel" code is not shown, as it is just several pages of error-checking.

Specific battle plan. I do not consider *all* the equivalences suggested by Figure 4.1; I focus only on the step-for-step equivalence of the λ_g- and λ_s-interpreters. Figure 4.2 suggests the comparison I will make—both interpreters take comparable action in each part of a reduction step: *search* for the "leftmost" redex (it may not literally be leftmost here), *copy* the rator if it is shared, do the *reduction*, and (optionally) *tidy* up the resulting λ-term.

$$\begin{tikzcd}
 & T^0_g\equiv T^0_t\equiv T^0_s \arrow[dl, "\text{onestepG}"'] \arrow[dr, "\text{onestepS}"] \arrow[dddd, bend right=90, "\text{onestepT*}"'] \arrow[dddd, bend left=90, "\text{eval\_WBNF}"] & \\
T^1_g \arrow[r, "\text{graph2termT}"] \arrow[d, "\text{onestepG}"'] & T^1_t & T^1_s \arrow[l, "\text{term2termT}"'] \arrow[ll, bend right, "\text{term2graph}"'] \arrow[d, "\text{onestepS}"] \\
T^2_g \arrow[r, "\text{graph2termT}"] \arrow[d, dotted, "\text{onestepG*}"'] & T^2_t & T^2_s \arrow[l, "\text{term2termT}"'] \arrow[ll, bend right, "\text{term2graph}"'] \arrow[d, dotted, "\text{onestepS*}"] \\
T^n_g \arrow[dr, "\text{graph2termT}"] & & T^n_s \arrow[dl, "\text{term2termT}"'] \arrow[ll, bend right, "\text{term2graph}"'] \\
 & T^n_t &
\end{tikzcd}$$

Figure 4.1: Graph reduction vs. reduction with suspensions

$$\begin{tikzcd}
T_g^{i} \arrow[d, "\textsc{Searching}"'] & T_s^{i} \arrow[l, "\mathsf{term2graph}"] \arrow[d] \\
T_g^{is} \arrow[d, "\textsc{Copying}"'] & T_s^{is} \arrow[d] \\
T_g^{ic} \arrow[d, "\textsc{Reducing}"'] & T_s^{ic} \arrow[l, "\mathsf{term2graph}"] \arrow[d] \\
T_g^{ir} \arrow[d, "\textsc{Tidying}"'] & T_s^{ir} \arrow[d] \\
T_g^{it} & T_s^{it} \arrow[l, "\mathsf{term2graph}"]
\end{tikzcd}$$

Figure 4.2: A λ_g-interpreter step vs. a λ_s-interpreter step

```
(* top-most loop: drives onestepS (the λs-interpreter). Does the skipping over top-level
   λs-abstractions.
*)

fun toplevS (Lam(B,n)) = Lam(toplevS, n) (* skipping over... *)
  | toplevS other      = real_toplevS other

and real_toplevS T =

let (* step forward *)
    val (action_in_T, T') = onestepS T
    (* tidy things up *)
    val T'' = trashpickup (tidyterm T')
in (* make sure all is well *)
    if not(is_well_formed T'') then (
        perr("malformed term! "^unparse(T''));
        T''
    ) else if action_in_T then (* keep going *)
        real_toplevS T''
    else
        T''
end
```

Notation: the output of reduction step i (T_g^{it} or T_s^{it}) is input to the next (T_g^{i+1} or T_s^{i+1}). (Note the s, c, r, and t superscripts for the four phases.)

Assuming a function onestepS (presented later) that does one reduction step, a λ_s-interpreter reduces a λ-term to WBNF by calling onestepS repeatedly, as the function toplevS : Term → Term (page 57) shows. It tidies terms with tidyterm and trashpickup; a call to is_well_formed is a safety check. (All these functions are presented later in this chapter.)

Representations. The functions formA2formB in Figure 4.1 indicate conversions between different representations of λ-terms; they will be introduced in due course. All the representations used (unadorned λ-trees without suspensions, λ_g-graphs, and λ-trees with suspensions) have the plain elements in common (applications, abstractions, constants and variables bound to abstractions). I routinely make comparisons between these common elements across representational boundaries, e.g., comparing a λ_g-abstraction to a λ_s-abstraction. Similarly, I will glibly call all of them "plain nodes."

After defining the data structures for the λ_s-interpreter, I examine how a λ_s-interpreter tackles the four phases of a reduction step. Because the phases interact in subtle ways, there is a chicken-and-egg problem with the presentation; if something seems murky the first time, skip it and try a second pass later.

4.1 λ_s-term structure and terminology

The λ_s-interpreter manipulates augmented λ-terms called *λ_s-terms*. λ_s-terms include the three plain elements of the λ-calculus: λ-applications, λ-abstractions, and variables bound to abstractions.

The new construct in λ_s-terms is the *suspension*. A suspension is a substitution put "on hold" or *suspended*.

The notation for a suspension is $[_x B\ P]$, and it stands for a substitution $B[x := P]$. Meanwhile all the instances of variable x in B are *sharing* the single copy of P. P is the called the *pointee* and B the *body* of the suspension. As always, the name x is a mnemonic decoration showing which variables in B are pointing to P. The only "cost" of removing *all* variable-names in λ_s-terms, leaving just binding indices, is in human convenience, as the example on page 21 showed.

In a λ_s-term, variables may be bound either to a λ_s-abstraction or to a suspension: both λ_s-abstractions and suspensions are binders (and have bound variables). A variable x_i bound to a suspension $[_x B\ P]$ "points to" or "is aimed at" a copy of the pointee P. I will call such suspension-bound variables *λ_s-pointers*. Henceforth, I reinforce the pointing notion by giving the pointers pointy hats, as in $\hat{x}_1$, $\acute{y}_3$, etc. (to be explained shortly). The best way to think about a suspension is as a peg on a λ_s-tree on which a shared λ_s-term is hung; variables "point" to the shared term with their binding indices.

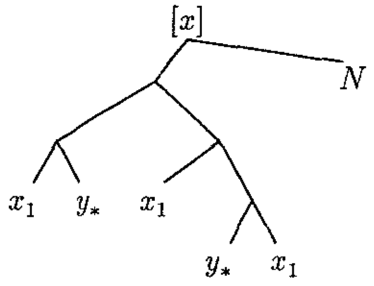
Figure 4.3 shows a λ_s-term with a suspension at its root. The three x_1 variables point to N.[1]

Figure 4.4 gives the ML definition for a well-formed λ_s-term; Figure 4.5 illustrates the various constructs (I will eventually account for all its dots and squiggles). A λ_s-term can be:

- An App(M,N): a λ_s-application; the rator M and the rand N must be well-formed λ_s-terms.

  As before, a λ_s-application is written as $(M\ N)$; in λ_s-tree form, it is an unmarked two-child node. Figure 4.5 has three λ_s-application nodes.

[1]Following my notational convention about asterisk subscripts, the y_* variables have some unknown and uninteresting binding indices. Also, recall that capital letters denote arbitrary terms.



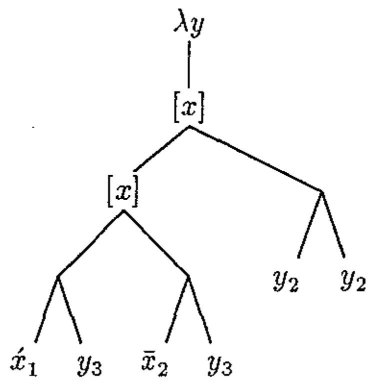
$$[_x((\hat{x}_1\ y_*)\ (\hat{x}_1\ (y_*\ \hat{x}_1)))\ N]$$

Figure 4.3: A λ_s-term with a suspension at its root

```
(* Terms in the pure λ-calculus, plus Sus(pensions) *)
datatype VarMark
    = NotPtr       (* plain variable; not a pointer *)
    | Ptr          (* var has been turned into a pointer *)
    | FollowFill   (* pointer being followed; should be filled eventually *)
    | FollowNoFill (* pointer being followed; needn't be filled *)
    | Followed     (* the following has been done [don't try again] *)

datatype Term                  (* std names used *)
  = App of Term * Term         (* M, N *)
  | Lam of Term * string       (* B, n *)
  | Var of int * VarMark * string (* bi, vmk, n *)
  | Sus of Term * Term * string   (* B, P, n *)
```

Figure 4.4: Definition of a Term



$$\lambda y.\{[_x[_x(\acute{x}_1\, y_3)\ (\bar{x}_2\, y_3)]\ (y_2\, y_2)]\}$$

Figure 4.5: An example with much λ_s-term notation

- A Sus(B,P,n): a suspension; the *body* B and the *pointee* P must be well-formed λ_s-terms. The suspension must not have a bound variable in P (Section 6.3 discusses lifting this restriction). n is a name, for decorative purposes only.

  A suspension is written as $[_n B\ P]$; in λ_s-tree form, it is shown by a $[n]$ node. Figure 4.5 shows two suspensions, both for variables decorated with the name x. Each suspension in the figure has one bound variable.

- A Lam(B,n): a λ_s-abstraction; the *body* B must be a well-formed λ_s-term. n is a variable name, for decorative purposes only.

  A λ_s-abstraction is written as $\lambda n.\{B\}$; in λ_s-tree form, it is shown by a λn node. The topmost node, λy, is the only Lam node in Figure 4.5.

- A Var(bi, vmk, n): a variable with binding index bi. bi < 0 means the variable is a constant (a variable free at the top level). The variable mark vmk is an annotation saying what has happened to the variable: NotPtr means it is a plain variable; it must have a λ_s-abstraction as its binder; a variable with any another mark must be bound to a suspension. The other variable marks (FollowFill, FollowNoFill, and Followed) are used to guide the following of λ_s-pointers. Section 4.3 describes their use. The variable name n is decorative, as always.

  A variable is written as n_{bi} in both tree and text form. "Hats" (accents) represent the variable marks. A NotPtr variable x_i has no hat; for Ptr,

```
(* is_well_formed : Term → bool.
   Uses chk_vars (page 170), is_ptr (page 169), and is_bd_var_or_ptr (page 172).
   A predicate function that says if a Term is "well formed." Mainly, it checks if variable
   bindings are well behaved. A λ-abstraction's bound variables must not be pointers, a
   suspension's bound variables must be pointers, and a suspension cannot have bound
   variables in its pointee.
*)
fun is_well_formed (Var(_,_,_))   = true
   | is_well_formed (App(M, N)) = (is_well_formed M) andalso (is_well_formed N)
   | is_well_formed (Lam(B, _))  =
     (is_well_formed B) andalso
     (if (bd_var_exists B) then (not(bd_vars_are_ptrs B)) else true)
   | is_well_formed (Sus(B, P, _)) =
     (is_well_formed B) andalso (is_well_formed P)
     andalso (if (bd_var_exists B) then (bd_vars_are_ptrs B) else true)
     andalso (not(bd_var_exists P))

and bd_var_exists T =
     chk_vars is_bd_var_or_ptr orElse false 1 1 T

and bd_vars_are_ptrs T =
 let fun bd_var_is_ptr lev _ (bi,vmk,n) = (bi=lev) andalso (is_ptr(Var(bi,vmk,n)))
     in chk_vars bd_var_is_ptr orElse false 1 1 T end
```

FollowFill, FollowNoFill, and Followed marks, I use $\hat{x}_i$, $\acute{x}_i$, $\grave{x}_i$, and $\bar{x}_i$ hats, respectively.

There are six variables in Figure 4.5. All the y's are NotPtrs (and have the same binder); the two x's are, left to right, a FollowFill and a Followed; they are bound to different suspensions.

The ML predicate function is_well_formed : Term → bool (page 61) encodes the requirements for a well-formed λ_s-term.

Converting between plain λ-terms and λ_s-terms. Plain (suspensionless) λ-terms *are* λ_s-terms.

A λ_s-term is converted to its *linear expansion* λ-term by completing the substitutions for which the suspensions in it stand, as the function term2termT : Term → Term (page 62) shows.

Notation for binding-index changes. I use the following notation for the adjustment of variables' binding indices:

```
(* term2termT : Term → Term.
   Converts a Term, possibly including suspensions, to one without suspensions (for
   pure tree reduction, hence the name "TermT"). Completes the substitutions that the
   suspensions represent.
   Uses std_subst (page 172) and incr_free_vars2 (page 171).
*)
fun term2termT (App(M,N))        = App(term2termT M, term2termT N)
  | term2termT (Lam(B, n))       = Lam(term2termT B, n)
  | term2termT (Var(bi,vmk,n)) = Var(bi,vmk,n)
  | term2termT (Sus(B,P, n))     =
    (incr_free_vars2  1 (std_subst (term2termT P) (term2termT B)))
```

T^{if}: Binding indices of free variables are incremented by 1; constants excluded.

T^{df}: Binding indices of free variables in T are decremented by 1; constants excluded.

T^{ib}: Binding indices of bound variables in T (the binder will be obvious from context) are incremented by 1; constants excluded.

T^{db}: Binding indices of bound variables are decremented by 1; constants excluded.

Terminology for comparing with λ_g-graphs. Recall that λ-applications, λ-abstractions, and variables bound to λ-abstractions are *plain nodes.* A λ_s-term, therefore, is made of plain nodes as well as suspensions and λ_s-pointers. If two plain nodes in a λ_s-term are *directly* connected by an edge in a λ_s-tree, they are g-connected (as in λ_g-graphs).

Two plain nodes in a λ_s-term may also be *s-connected*, with an "indirection" through suspensions and/or a λ_s-pointer. As will only become clear in Section 4.5.5, the tidying rules (Section 4.5) guarantee that s-connections must have one of two forms in a tidied λ_s-term. Meanwhile, the more intuitive notions presented shortly will get us through.

If two plain nodes in a λ_s-term are either g- or s-connected, then they are *sg-connected.* (A λ_g-graph is trivially sg-connected, because all of its nodes are plain, and adjacent nodes are invariably g-connected.) I re-emphasize that all this "connecting" has only to do with *plain* nodes in λ_s-terms (and λ_g-graphs, of course).

A tidied λ_s-term T_s is *λ_{sg}-equivalent* to a λ_g-graph T_g (written as $T_s \equiv_{sg} T_g$) if T_s may be made λ_g-equivalent to T_g by converting all the s-connections in T_s to g-connections. Put colloquially and none too precisely: "Replace the λ_s-pointers with real pointers and throw away the suspensions." Algorithm 4.1 (page 64) will be a more careful statement of this idea.

4.2 β_s-reduction: the β_s rule

This section examines how the λ_s-interpreter does β-reduction. Even though finding redexes (Section 4.3) and copying shared rators (Section 4.4) come first in a reduction step, looking at β-reduction helps to clarify those more involved tasks.

As Figure 4.2 showed, the λ_s-term input to this phase is T_s^{ic} and its λ_{sg}-equivalent is T_g^{ic}. (Sections 4.3 and 4.4 will confirm that the searching and copying phases preserve λ_{sg}-equivalence.) The reduction phase's output will be T_s^{ir} and T_g^{ir}, respectively.

T_s^{ic} is a tidy λ_s-term; for this section, that just means that the root of the redex is g-connected to its rator. (Section 4.5 defines tidying more generally and discusses it at some length). Earlier parts of a reduction step do not affect tidiness: searching for a redex does not alter terms and copying a shared rator preserves tidiness (Lemma 4.8).

β_s-reduction (the *β_s rule*) in the λ_s-interpreter creates a suspension:

$$(\lambda x.\{B\}\ P) \to [_x B\ P^{if}].$$

Binding indices of free variables in P must be incremented because a new binder has been thrown into their binding paths, and the variables bound by the λx must have their vmk's changed to Ptr because their binder is now a suspension. Sample code for the β_s rule is halfway down the code for onestepS (page 72).

Lemma 4.1 *Given $T_s^{\text{ic}} \equiv_{sg} T_g^{\text{ic}}$ with corresponding redexes selected for β-reduction, if T_s^{ic} is β_s-reduced to T_s^{ir} and T_g^{ic} is β_g-reduced to T_g^{ir}, then $T_s^{\text{ir}} \equiv_{sg} T_g^{\text{ir}}$.*

Proof. The proof is by showing that the interpreters make corresponding changes to their connected plain nodes and to the connections. There are three special cases to consider: when the rator body includes no bound variables, when the rator body is a single bound variable, and when the rand is

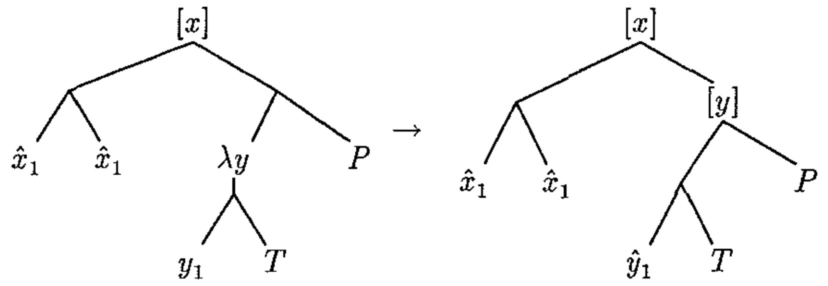


Figure 4.6: β_s-reduction of a shared redex

a single variable. These cases are inextricably linked with tidying, specifically the [useless] rule (Section 4.5.1) and trivial-suspension eradication (Section 4.5.2); they are dealt with there. Lemmas 4.3, 4.4, and 4.5 guarantee the correctness (corresponding plain nodes and connections) of those cases.

Table 4.1 carefully compares the λ_g-interpreter and λ_s-interpreter implementations of β-reduction for all cases other than those just mentioned. It shows that the correspondence between plain nodes and their connections is maintained. □

The blow-by-blow comparison of β_g- and β_s-reduction suggests an algorithm to convert a λ_s-term to a λ_g-equivalent λ_g-graph.

Algorithm 4.1 Convert a tidied λ_s-term to a λ_g-graph. In short, replace each s-connection between plain nodes with a g-connection.

1. Plain nodes in the λ_s-term carry over unchanged, as do any g-connections (edges) directly connecting them.

2. Each λ_s-pointer is replaced with a "real" pointer to the root node of its binder's pointee; tidying ensures that the root node is a plain node.

3. Flag the plain root node of the pointee as a SFE (in λ_g-interpreter ML code, set `subbed := true`, as in `onestepG` (page 36)).

4. "Short-circuit" the suspension node by replacing any pointers to it with pointers to its body.

5. Give each variable bound to a λ-abstraction a Wadsworth-style back-pointer that points to the binder indicated by its binding index. (Recall from Section 3.4.1 that binding indices do not work in λ_g-graphs).

	λ_g-interpreter	λ_s-interpreter
1.	Replaces bound variables with pointers to the rand: g-connections.	Marks bound variables as "pointing" to the rand (now a suspension's pointee): s-connections.
2.	Keeps (shares) only one copy of the rand.	Keeps one copy of the rand as the suspension's pointee.
3.	Marks the rand as a substituted free expression (SFE) (subbed flag set to true).	The root node of the rand can be distinguished as "substituted" because it is the root of a suspension's pointee.
4.	The root node of the redex is overwritten with the root node of the result, so that if the redex was shared, the sharers will all see the reduced version. The redex-root λ_g-application node is thus effectively discarded.	The root node of the redex is effectively overwritten with the root node of the result by turning the redex into a suspension. A redex is shared if it occurs in the pointee of *another* suspension, as in Figure 4.6. Because the redex-root λ_s-application node is replaced by a new suspension, all the λ_s-pointers that used to point to the redex now point to its reduced form (the new suspension).
5.	The rator-root λ_g-abstraction node is discarded.	The rator-root λ_s-abstraction node is discarded.
6.	Plain nodes lost are the redex-root, the rator-root, and all the rator-bound variables.	Plain nodes lost are the redex-root (it becomes a suspension), the rator-root, and all the rator-bound variables, which become λ_s-pointers.

Table 4.1: Comparison of β_g- and β_s-reduction

All the conversions in Algorithm 4.1 (real pointers for λ_s-pointers, "short-circuit wires" for suspensions, and backpointers for λ-abstraction-bound variables) are independent of each other, so the λ_g-graph produced does not depend on the order in which the conversions are done. Term2graph : Term $\rightarrow$ Gnode ref (page 67) is an implementation of the algorithm.

4.3 Searching for the next redex

For comparison purposes, the inputs to this phase are T_s^i and T_g^i, with outputs T_s^{is} and T_g^{is}, respectively, as in the battle plan, Figure 4.2.

Following pointers. Evaluation to WBNF of a λ-term repeatedly seeks out the leftmost redex not in a top-level λ-abstraction and reduces it. In trees or acyclic graphs, the first redex encountered in a pre-order walk from the root is selected. For ordinary tree reduction or the λ_g-interpreter, a tree- or graph-walking algorithm that follows g-connections between plain nodes suffices, as the functions onestepT (page 23) and onestepG (page 36) exemplify.

How does one do a pre-order walk to the next redex in a λ_s-tree with suspensions and λ_s-pointers? It should be no surprise that the principle is to follow s-connections in the same way as the usual g-connections. Consider Figure 4.7, which shows a term after one β_g- and β_s-reduction. The next redex—the only one—is $(\lambda x.\{x\}\ y)$, $\star$'d in the figure. Graph reduction will find it with a preorder walk (call it a λ_g-walk), visiting nodes λx, λy, (), (), x, λa, (), a, () [redex found].

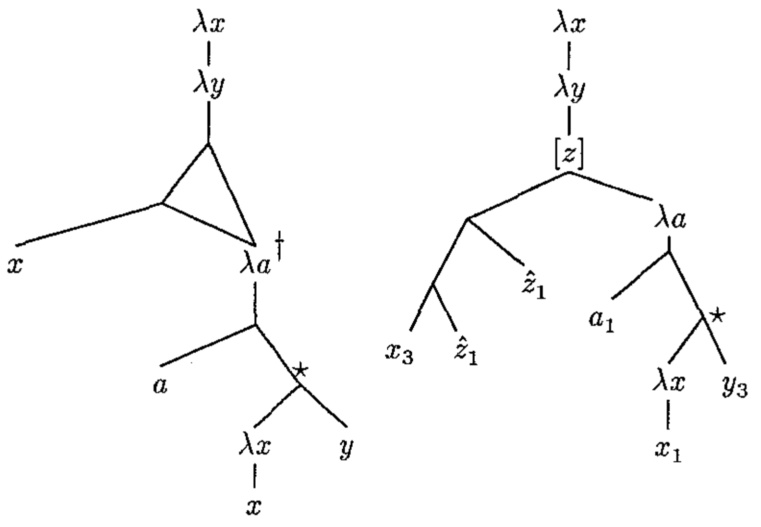
What does a pre-order walk of the comparable λ_s-term give (a λ_s-walk)? The first part is: λx, λy, $[z]$, (), (), x_3, $\hat{z}_1$. We observe:

- There is a suspension node ($[z]$) in the λ_s-walk but not in the λ_g-walk. This absence fits with my earlier suggestion that a suspension is just a "peg" on the tree on which a shared term (the pointee) has been hung. When on a redex-finding mission, we should ignore the pegs insofar as possible.

- The new aspect of the λ_s-walk is running into the $\hat{z}_1$. What does that mean? At the equivalent point in the λ_g-walk, we *followed a pointer* from a λ_s-application () to its rand λa. But what *is* $\hat{z}$? It is a *λ_s-pointer* to its binder's pointee! Follow it.

```
(* The functions term2graph : Term → Gnode ref  and graph2termT : Gnode ref → Term
   convert between λs-terms and λg-graphs.
   term2graph uses next_ID (page 177), bidx_to_bndrIDs_T (page 177), incr_refcnt (page
   175), set_subbed (page 175), and substG (page 37). graph2termT uses bndrIDs_to_bidx
   (page 177).
*)
and term2graph (App(M,N)) =
     ref(AppG((term2graph M), (term2graph N), ref false,
          (ref false, ref 1, ref false, ref (0,0))))
   | term2graph (Sus(M,N, n)) =
     (* finish the substitution [innermost out] *)
     let val id_to_use              = next_ID ()
          val Mg                    = term2graph (bidx_to_bndrIDs_T 1 id_to_use M)
          val Ng                    = term2graph (bidx_to_bndrIDs_T 1 id_to_use N)
          val _                     = incr_refcnt ~1 Ng;
          val _                     = set_subbed true Ng;
          val (substituted, no_subs) = (substG id_to_use Ng Mg)
     in substituted end
   | term2graph (Lam(B, n)) =
     let val id_to_use = next_ID ()
          val Bt'       = bidx_to_bndrIDs_T 1 id_to_use B
          val B'        = term2graph Bt'
     in ref(LamG(B', ref id_to_use, n, (ref false, ref 1, ref false, ref (0,0)))) end
   | term2graph (Var(bi,vmk,n)) = (* VarMark info thrown away *)
     ref(VarG(ref bi, n, (ref false, ref 1, ref false, ref (0,0))))

and graph2termT (G as ref (AppG(M,N,indir,_))) =
     if !indir then (* indirection node *)
          graph2termT M
     else
          App(graph2termT M, graph2termT N)
   | graph2termT (ref(LamG(B,si,n,_))) =
     Lam(bndrIDs_to_bidx 1 (!si) (graph2termT B), n)
   | graph2termT (ref(VarG(si,n,_))) =
     Var((!si), NotPtr, n) (* NB: put binderID in temporarily *)
```



(a) λ_g-graph (b) λ_s-term

Figure 4.7: A term after one β_g- or β_s-reduction

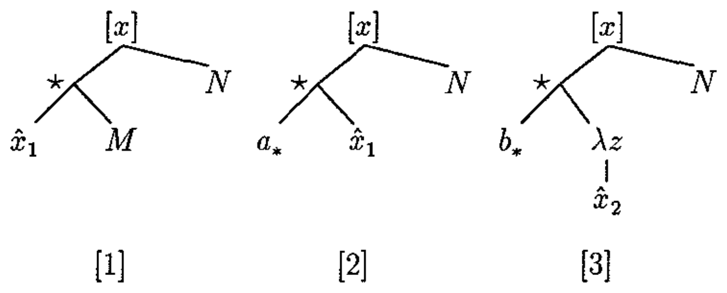
Therefore, to mimic graph reduction, we follow $\hat{z}_1$ by re-visiting its binder and continuing the preorder walk at its pointee. The λ_s-walk would continue: $[z]$, λa, $()$, a_1, $()$ [redex found].

(Alternatively, instead of following the λ_s-pointer to its target term, we could bring the term to the λ_s-pointer—that is, copy the term. Though this would make for a kind of "lazy" tree reduction, it would not be anything like graph reduction.)

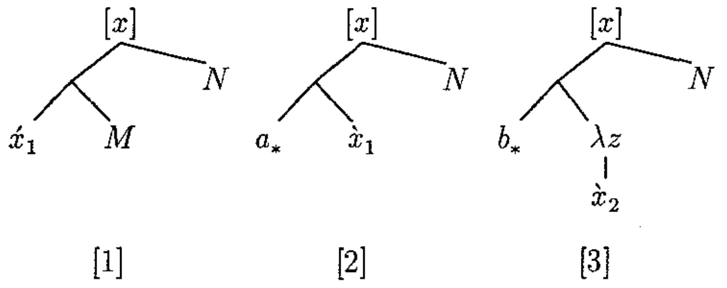
What if the walk down the pointee finds no redex? Where should the search resume? Again, the cue comes from graph reduction: the search resumes at the pointer that carried us off in the first place.

What if the walk down the pointee runs into *another* λ_s-pointer? (Well-formedness prevents it from taking the search to an already-searched binder.) We follow it, as before.

Running into λ_s-pointers: setting variable-marks. Searching for the next redex in a λ_s-term by doing a pre-order walk that follows both g- and s-connections is not a difficult idea. I put it into practice in the form of a system that manipulates λ_s-pointers' hats; the system is mainly driven by the needs of the FFPM implementation in Chapter 5.



(a) three λ_s-pointers to be marked



(b) three λ_s-pointers, marked

Figure 4.8: The three cases of pointer-following

If a λ_s-interpreter is looking at a suspension, it needs to know "where the action is." Is a λ_s-pointer to some suspension higher up in the λ_s-tree being followed? Is a λ_s-pointer to *this* suspension being followed? Have *all* the λ_s-pointers bound to this suspension already been followed? These kinds of questions can be answered by the judicious changing of hats.

Consider Figure 4.8a, which illustrates the only three ways a λ_s-pointer can be a child node: as rand and rator of a λ_s-application, or as the body of a λ_s-abstraction. (The case of a λ_s-pointer as a suspension's child is untidy and therefore disallowed, as discussed in Section 4.5.2.) In all three cases, assume we are continuing a tree-walk from the $\star$'d λ_s-application node.

In Figure 4.8a[1], the λ_s-application's rator is the pointer $\hat{x}_1$. If $\hat{x}_1$ points to a λ_s-abstraction (i.e., N is one), then we need a copy of it here, because *this* λ_s-application is the next redex! On the other hand, if N is not a λ_s-abstraction, then we want to look for a redex in it, in hopes that it will become a λ_s-abstraction. (This is analogous to the function of eval_LF in the eval-apply interpreter eval_BNF (page 25).) If we eventually run out of redexes in N, then we want to return to the λ_s-pointer that started the search at the pointee.

What actually happens when the λ_s-interpreter reaches the rator λ_s-pointer? The λ_s-interpreter marks x_1 with a FollowFill mark, indicating that it decided to follow the pointer and that it should be filled (replaced by a copy) if the pointee is or becomes a λ_s-abstraction. The hat on the marked $\acute{x}_1$ slants the same way as a rator slants off a λ_s-application node.

In Figure 4.8a[2], the rator of the λ_s-application is a constant (it could be any term without a redex or unFollowed λ_s-pointer), so the tree-walk search for a redex goes on to examine the rand, a pointer $\hat{x}_1$. No matter what $\hat{x}_1$ points to, the λ_s-application is not going to be a redex. There is no reason to fill this pointer. So, we mark x_1 with a FollowNoFill mark (shown with a rand-ward slant accent: $\grave{x}_1$). This means that $\grave{x}_1$ should be followed and eventually returned to, but never copied into.

In Figure 4.8a[3], the rator of the λ_s-application is again a constant (or any term without a redex or unFollowed λ_s-pointer), and the rand is a λ_s-abstraction, so the tree-walk continues into the abstraction's body, where lurks the λ_s-pointer $\hat{x}_2$. Again, no matter what $\hat{x}_2$ points to, there will be no redex here. Therefore, here also, the variable will be marked FollowNoFill: $\grave{x}_2$.

Figure 4.8b shows how the three cases would be marked so $\hat{x}_i$ could be followed effectively.

The life cycle of a λ_s-pointer. Now that most of the λ_s-pointer hats have been introduced, we can consider how they are generally used. For this, think of λ_s-terms in their linear, textual representation.

Before the λ_s-pointer, there is a variable bound to a λ_s-abstraction, with a NotPtr hat. When a β_s-reduction takes place, the λ_s-pointer is born, with a Ptr hat. If and when the search for redexes reaches its part of the term, a λ_s-pointer may get a FollowFill or FollowNoFill hat. Following one λ_s-pointer may spark the need to follow another further *to the right* in the term. As the following yields results (target term is reduced, target term is a λ_s-abstraction, etc.), λ_s-pointers will be marked as Followed, with flat-top hats $\bar{x}_i$, meaning it is useless to re-follow them. *Very* roughly, the distribution of λ_s-pointers in a linearly-represented λ_s-term will be that in Figure 4.9. In the beginning, all variables will be NotPtrs, and in the end there will be many Followed λ_s-pointers. In between, a "wave" of Ptr and Follow-type pointers will sweep left-to-right as the leftmost redex becomes more and more to the right.

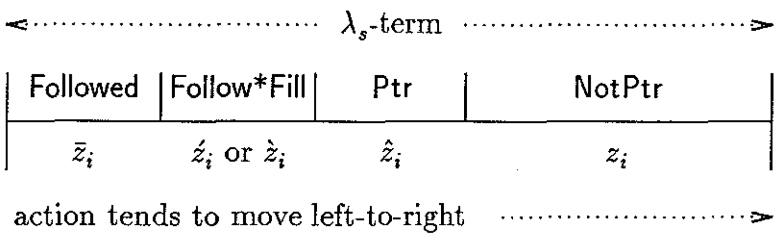


Figure 4.9: Example distribution of λ_s-pointers in a λ_s-term

Figure 4.9 reflects a *general* pattern of reduction and is *only* a visual image that may benefit some readers. Examples with differently-hatted λ_s-pointers intermingled in arbitrary ways are easy to make up.

An ML implementation of searching for a redex. Implementations of the basic idea of following s-connections as well as g-connections may differ dramatically in how they work. I offer as evidence the ML implementation in this chapter, which uses an ML data type to represent λ_s-trees, and the FFPM implementation in Chapter 5, which operates on a linear string of symbols.

The function onestepS : Term $\rightarrow$ Term (beginning on page 72) is the heart of my λ_s-interpreter implementation in ML; it encodes β_s-reduction and the search for a redex, with its pointer hats, following, filling, etc. Each time onestepS is called, it looks for a β_s-redex, reduces it, reports its success, and returns the new λ_s-term. Complexity rears its ugly head at a suspension, on the second page of onestepS. The review here recaps some of the preceding exposition.

If onestepS reaches a suspension in its walk, it should search the suspension's body. It should not search the pointee (it should walk past the peg on the tree) *unless* the suspension has a bound variable with an appropriate Follow-type mark.

If the suspension has a bound variable with a FollowFill or FollowNoFill hat, then attention focuses on the suspension's pointee. (In normal-order evaluation, redex detection is serial so a suspension will have at most one of these Follow-marked bound variables to obey.)

The easiest case is with a FollowNoFill bound variable; the tree walk should simply continue with the pointee. The pointee should be reduced to β-normal form (BNF), and (assuming termination) the FollowNoFill hat should be changed to Followed.

```
(* onestepS : Term → (bool,Term).
   Uses ptrize_bd_vars (page 173), incr_free_vars1 (page 171), chk_vars (page 170),
   mod_vars (page 171), is_bd_follow_ptr (page 172), is_bd_follow_fill_ptr (page 172),
   is_higher_up_follow_ptr (page 172), and subst (page 172).
   Given a Term, find first redex in it and reduce it. Report whether or not a reduction
   was done (and return new term). Reduces to weak β-normal form (WBNF).
*)
fun onestepS (Lam(B,n)) = (* easy ones first ... *)
    (false, Lam(B,n))
(* if this were for β-normal form (BNF)...
  — onestepS (Lam(B,n)) =
    let val (done_in_B, B') = onestepS B
    in (done_in_B, Lam(B', n)) end
*)
  | onestepS (Var(x,Ptr, n)) = (false, Var(x,FollowNoFill,n))
  | onestepS (Var(x,FollowFill, n)) = raise follow_ptr_during_search_error
  | onestepS (Var(x,FollowNoFill, n)) = raise follow_ptr_during_search_error
  | onestepS (Var(x,vmk, n)) = (false, Var(x,vmk,n))
  | onestepS (App(Lam(B,n),N)) = (* fire the β_s rule! *)
    (true, ptrize_bd_vars (Sus(B, (incr_free_vars1 1 N), n)))
  | onestepS (App(Var(x,Ptr,n),N)) = (* rator is a λ_s-pointer *)
    (false, (App(Var(x,FollowFill,n),N)))
  | onestepS (App(M,N)) = (* rator something else *)
    let val (done_in_M, M') = onestepS M
        fun higher_up_follow_ptr_exists T =
            chk_vars is_higher_up_follow_ptr orElse false 0 0 T
    in if done_in_M then
            (true, App(M',N))
        else if higher_up_follow_ptr_exists M' then
            (false, App(M',N))
        else (* truly nothing happened in M *)
        let val (done_in_N, N') = onestepS N
        in (done_in_N, App(M',N')) end
    end
 (* onestepS continues on the next page. *)
(* This is a continuation of onestepS; this part handles suspensions. *)
  | onestepS (Sus(B,P, n)) = (* define local functions first *)
   let fun bd_follow_fill_ptr_exists T =
           chk_vars is_bd_follow_fill_ptr orElse false 1 1 T

       fun bd_follow_ptr_exists T =
           chk_vars is_bd_follow_ptr orElse false 1 1 T

       fun higher_up_follow_ptr_exists T =
           chk_vars is_higher_up_follow_ptr orElse false 1 1 T

       fun mk_bd_followed_ptrs T =
       let fun make_followed_ptr lev _ (db,FollowFill,n) = (db, Followed, n)
             | make_followed_ptr lev _ (db,FollowNoFill,n) = (db, Followed, n)
             | make_followed_ptr lev _ ( _, _,_) = raise ptrize_bd_vars_error
       in mod_vars is_bd_follow_ptr make_followed_ptr 1 1 T end
   in
       if (bd_follow_fill_ptr_exists B andalso is_lam P) then
           let val B' = (subst is_bd_follow_fill_ptr 1 1 P B) (* fill 'er up *)
               val (done_in_B',B") = onestepS B' (* do the redex just created *)
           in (true, Sus(B", P, n)) end
       else if (bd_follow_ptr_exists B) (* but P not a λ-abstraction *) then
           let
               val (done_in_P, P') = onestepS P in (* try for a redex in P *)
               if done_in_P then
                   (true, Sus(B, P', n))
               else if higher_up_follow_ptr_exists P' then (* hit another λ_s-ptr *)
                   (false, Sus(B, P', n))
               else (* reduced; re-mark pointers; try again in body B *)
                   let val B' = mk_bd_followed_ptrs B
                       val (done_in_B', B") = onestepS B'
                   in (done_in_B', Sus(B",P, n)) end
           end
       else (* no Follow pointer in body B; try for a redex in there *)
           let
               val (done_in_B, B') = onestepS B in
               if done_in_B then (* great, something happened *)
                   (true, Sus(B', P, n))
               else if (bd_follow_ptr_exists B') then (* a Follow ptr was made *)
                   onestepS (Sus(B', P, n))
               else (* λ_s-ptr upward hit, or finished *)
                   (false, Sus(B', P, n))
           end
  end
```

If a FollowFill bound variable exists in a suspension's body, then a λ_s-application down there needs a λ_s-abstraction rator. If the pointee is a λ_s-abstraction, then the next redex has been found, but the rator is not "in place." Section 4.4 is concerned with the copy (or move) that must precede the β_s-reduction.

If the target of a FollowFill λ_s-pointer is not a λ_s-abstraction, onestepS looks for redexes in the pointee in the usual tree-walking way. However, it stops looking if the pointee becomes a λ_s-abstraction (and does the single substitution as before). This is evaluation to *Root-lambda form* (RLF): reduction proceeds until the root node of the λ_s-term becomes a λ_s-abstraction or until BNF is reached, whichever comes first. RLF is tied to overwriting of redexes' root nodes; it is analogous to lambda form (LF) attempted by the auxiliary function eval_LF of the eval-apply interpreter eval_BNF (page 25).

If, in dealing with a FollowFill pointer, the suspension's pointee is not a λ_s-abstraction and does not become one, then the FollowFill pointer should be changed to Followed and the search for a redex should resume in the suspension body. When re-hatting, all of the suspension's bound variables can be changed.

Further searching for redexes in a suspension's subordinate λ_s-terms is likely to run into *another* λ_s-pointer, perhaps one with a target suspension higher up in the overall λ_s-tree. In the λ_s-interpreter, active work is done on the λ_s-pointer poking highest up in the tree, and work on the others is deferred.

Searching for a redex: summary. Because T^i_s and T^i_g have the same sets of sg-connected plain nodes and the specification for the search is a pre-order walk that follows s-connections as well as g-connections, corresponding plain nodes will be chosen as the next redex (if one exists).

The only changes made to T^i_s while searching for a redex are changed hats on λ_s-pointers. These changes do not affect λ_{sg}-equivalence (Algorithm 4.1 pays no attention to variables' hats), so if $T^i_s \equiv_{sg} T^i_g$ before searching started, they will also be when searching is finished.

The tidiness of T^i_s must also be preserved, because searching does not change the λ_s-term's structure.

4.4 Lazy copying of shared rators

For comparison purposes, the inputs to this phase are T_s^{is} and T_g^{is}, with outputs T_s^{ic} and T_g^{ic}, respectively, as the battle-plan showed (Figure 4.2).

If the λ-abstraction rator of a redex is shared, some or all of it must be copied to avoid "using up" the λ-abstraction template. Section 3.3 introduced this idea and explained "lazy" (vs. "fully lazy") copying. A lazy copy is one that shares SFEs, those known to be free (because they were substituted into a λ-abstraction from above and cannot, therefore, include any variables bound to that abstraction). A fully-lazy copy goes further, seeking out maximal free expressions (MFEs) at every step.

In the λ_s-interpreter, a λ_s-term of the form $(\acute{x}_i\ R)$, where $\acute{x}_i$ is aimed at a λ_s-abstraction, is a redex. The λ_s-pointer x_i needs to be filled (that is why it has a FollowFill hat); there are two cases. If there is more than one λ_s-pointer pointing to the λ_s-abstraction rator, then it is *shared* and must be copied. I will show that the corresponding plain nodes are copied as in graph reduction and that the same connections will be set up.

The second case is when the $\acute{x}_i$ λ_s-pointer is the only one aimed at the λ_s-abstraction rator. It still needs to replace $\acute{x}_i$, so all parts of the redex will be local. After the replacement the suspension node and its pointee may be removed (they are now useless); conceptually, the λ_s-abstraction is being moved. I call this case a *last-instance relocation*, because the last copy of the λ_s-abstraction is being moved to where it will be used. This relocation does not affect λ_{sg}-equivalence because a set of nodes is simply being moved from one place to another, and the upward s-connection from redex to rator is being replaced with a g-connection. The λ_s-abstraction node loses its SFE-ness (detectable in a λ_s-term by being the pointee of a suspension), but this node is thrown away in the immediately-following β_s-reduction, so the λ_g/λ_sdiscrepancy goes away.

Graph reduction has no counterpart to last-instance relocation; Section 5.3.2 (page 145) discusses the matter at some length, because of its effect on a λ_s-interpreter's time complexity.

When a graph reducer copies a shared rator lazily, it copies one or more plain, non-SFE, g-connected nodes that have a "border" of SFE root nodes. The new copy will share the SFE border nodes with the original copy.

A lazy copy in the λ_s-interpreter simply copies the λ_s-abstraction pointee of the suspension at which the FollowFill λ_s-pointer is aimed. Figure 4.10 shows an example of lazy copying in the λ_s-interpreter similar to that for the λ_g-interpreter shown in Figure 3.5 (page 39). The figures show (a) just

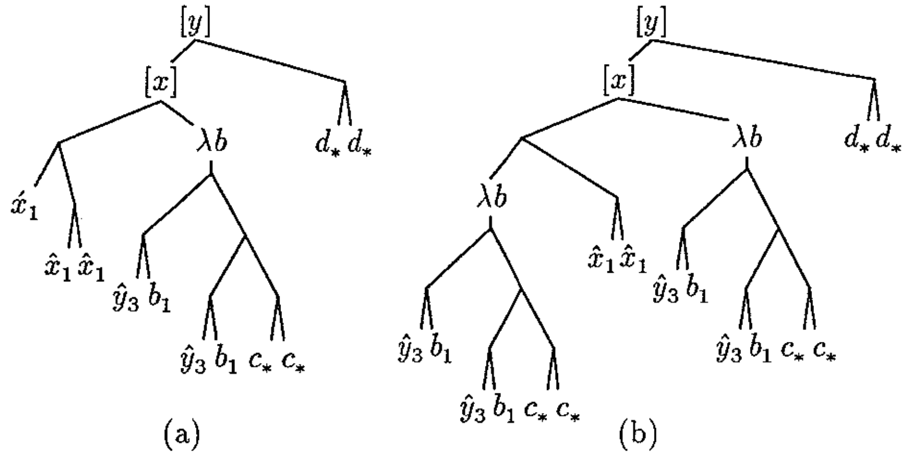


Figure 4.10: A lazy copy in the λ_s-interpreter

before the lazy copy and (b) just after.

Lemma 4.2 *The lazy copying of a shared λ-abstraction rator in a λ_g-interpreter and a λ_s-interpreter maintains the correspondence between plain nodes and their connectedness; that is, if* $T_s^{\text{is}} \underset{sg}{\equiv} T_g^{\text{is}}$ *then* $T_s^{\text{ic}} \underset{sg}{\equiv} T_g^{\text{ic}}$.

Proof. Table 4.2 gives a blow-by-blow comparison of λ_g- and λ_s-interpreter operations and shows that the plain-nodes and connectedness equivalence is maintained. The ML code for lazy_copy (page 41) may also be instructive.

The λ_s-interpreter part of step 3 in Table 4.2 deals with nicely g-connected plain nodes in the suspension's pointee and with λ_s-pointers that are upward s-connections to pointees higher up in the λ_s-tree. Fortunately, there are no other s-connections or λ_s-pointers *within* the pointee because it cannot have a suspension in it. A suspension in the pointee would indicate a β_s-reduction was done inside a λ_s-abstraction, which reduction to WBNF prevents. □

Comment. Because the λ_s-interpreter does lazy copies, if λ-lifted terms were provided as input, then, according to Arvind et al.'s result [9], it would give the same sharing as Wadsworth's fully-lazy-copying interpreter.

4.5 Tidying λ_s-terms

Besides a search strategy to find redexes, a method for copying shared rators, and a β-reduction rule, a λ-calculus interpreter may have a tidying phase that

	λ_g-interpreter	λ_s-interpreter
1.	Copy the λ_g-abstraction root-node (a plain node). Consider copying its body (Step 2).	Copy the λ_s-abstraction root-node (a plain node). Consider copying its body (Step 2).
2.	If a node is the root of a SFE, do not copy it but g-connect its parent to node itself.	The corresponding thing to a g-connection to a SFE is a λ_s-pointer aimed at a suspension's pointee (an upward s-connection). Copy the λ_s-pointer, creating an upward s-connection to the same pointee.
3.	If a (plain) node is not the root of a SFE, copy it and g-connect it to its copied parent.	Copy the corresponding plain node and g-connect it to its copied parent.

Table 4.2: Comparison of λ_g- and λ_s-lazy copying

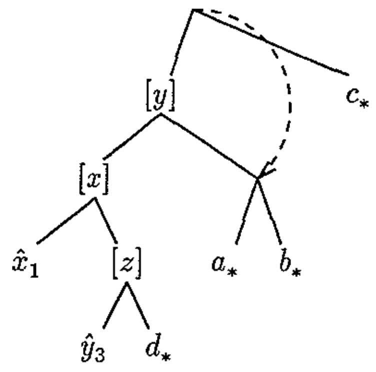


Figure 4.11: Pointer-following run amok

rearranges terms to some advantage. The removal of indirection nodes is an example,[2] though most graph reducers avoid such overhead. I ignore tidying for λ_g-graphs.

Following the battle plan (Figure 4.2), the inputs to this phase are T_s^{ir} and T_g^{ir}, with outputs T_s^{it} and T_g^{it}, respectively. Because λ_g-graphs are not tidied and this is the last phase of reduction-step i, $T_s^{it} = T_s^{i+1}$ and $T_g^{ir} = T_g^{it} = T_g^{i+1}$.

Tidying up λ_s-terms between β_s-reduction steps is important for a λ_s-interpreter. The reason is because unrestrained λ_s-pointer-following can quickly get out of hand; Figure 4.11 shows why. Assuming that we begin looking for a redex at the root node in the figure, the dotted arc shows the next plain node that we will get to. Whereas graph reduction will make the hop in one step (probably with one machine instruction), λ_s-pointer-following will visit (non-plain) nodes $\hat{x}_1$, $[x]$, $\hat{y}_3$, and $[y]$ (among others) before reaching the plain-node pointer target. One may put together trees of suspensions and λ_s-pointers to create arbitrarily complicated hops from one plain node to another.

This section introduces the *tidying rules* for the λ_s-interpreter designed to improve λ_s-pointer-following by rearranging suspensions. Most of the section is in a tutorial style; Section 4.5.7 summarizes the tidying enterprise. At the λ-calculus level, these rules are identities related to the substitution operation; I rely on the presentation about substitution in Hindley and Seldin's text [96, pages 7–10].

The first tidying rules are intimately related to the special cases of β-reduction in the λ_g-interpreter, described in Section 3.2.

[2]The function rm_indir_nodes (page 176) is an implementation.

```
(* trashpickup : Term → Term.  Suspensions without bound variables are removed.
   Uses chk_vars (page 170), is_bd_var_or_ptr (page 172), and incr_free_vars2 (page 171).
*)
fun trashpickup (App(M, N))       = App(trashpickup M, trashpickup N)
  | trashpickup (Lam(B, n))       = Lam(trashpickup B, n)
  | trashpickup (Var(bi,vmk, n))  = Var(bi,vmk,n)
  | trashpickup (Sus(B, P, n))    =
   let val B' = trashpickup B
       val P' = trashpickup P
       val bd_var_in_body = (chk_vars is_bd_var_or_ptr orElse false 1 1 B')
   in
       if not(bd_var_in_body) then incr_free_vars2 ~1 B'
       else Sus(B', P', n)
   end
```

4.5.1 Removing useless suspensions

A *useless suspension* is one with no variables bound to it; no λ_s-pointers in its body are aimed at its pointee. This is garbage collected in LISP and other systems with automatic storage management. Useless suspensions are the reason I refer to "*connected* plain nodes," because useless suspensions can add arbitrarily many (unconnected) plain nodes to a λ_s-term.

A suspension becomes useless either because it had no bound variables when it was created (β_s rule) or because its initially-bound variables have all been filled in subsequently. Removing a useless suspension is not a reduction rule in the pure sense; however for convenience, I will lump it with the more proper rules to be applied when tidying. This pseudo-rule is:

$$\text{[useless]:} \quad [B\ P] \quad \to \quad B^{df}, \quad B \text{ contains no bound variables.}$$

The [useless] pseudo-rule is just Hindley and Seldin's Lemma 1.14b:

$$\text{if } x \notin \mathit{FreeVars}(M) \text{ then } M[x := N] \equiv M.$$

My ML implementation scans for useless suspensions after each step, using the function trashpickup : Term → Term (page 79). The FFPM implementation in Chapter 5 checks for them immediately after β_s-reductions and the filling of FollowFill λ_s-pointers.

Lemma 4.3 *A β_g-reduction $T_g^{\text{ic}} \to T_g^{\text{ir}}$ in which the rator has no bound variables corresponds to a β_s-reduction $T_s^{\text{ic}} \to T_s^{\text{ir}}$ followed by an application of* *the* [useless] *pseudo-rule.*

Proof. It is straightforward:

$$\begin{array}{lllllll} g \text{ case:} & (\lambda x.\{M\}\ N) & & & \to & M & \\ s \text{ case:} & (\lambda x.\{M\}\ N) & \to & [_x M\ N] & \to & M & \square \end{array}$$

4.5.2 Removing trivial suspensions

One cause of needlessly painful λ_s-pointer-following is *trivial suspensions*: those that have a single pointer as their body or pointee. A trivial suspension wastes space and provides no sharing. It provides no benefit, so it can be removed. There are two rules:

[triv-body]:	$[x_1\ N]$	$\to$	N^{df}	NB: cases in which the single-variable body is not bound to the suspension are covered by the [useless] pseudo-rule.
[triv-ptee]:	$[M\ y_i]$	$\to$	M'^{df}	M' is M with bound variables reset to i; then decrement all the free variables (including y_i).

The [triv-body] rule applies if the *body* of a suspension is a lone *bound* variable; the (trivial) substitution may be completed. If the variable is not bound here, the [useless] pseudo-rule applies.

The [triv-ptee] rule applies if the *pointee* of a suspension is a lone variable. It is best to complete the substitution $M[x := y_i]$—variables replace variables, and an unnecessary indirection is removed.

For both rules, free variables must be decremented when the suspension is removed, because one binder has been removed from the variables' binding paths.

Lemma 4.4 *A β_g-reduction $\underset{g}{T^{\text{ic}}} \to \underset{g}{T^{\text{ir}}}$ in which the rator is a lone variable corresponds to a β_s-reduction $\underset{s}{T^{\text{ic}}} \to \underset{s}{T^{\text{ir}}}$ followed by an application of the* [triv-body] *rule.*

Proof. The λ_g-interpreter treats this case specially by setting the rand's subbed flag to false, even if it was set previously. This keeps λ_{sg}-equivalence from being broken by a single subbed flag! Consider the two cases, with a dagger superscript indicating the subbed flag is set:

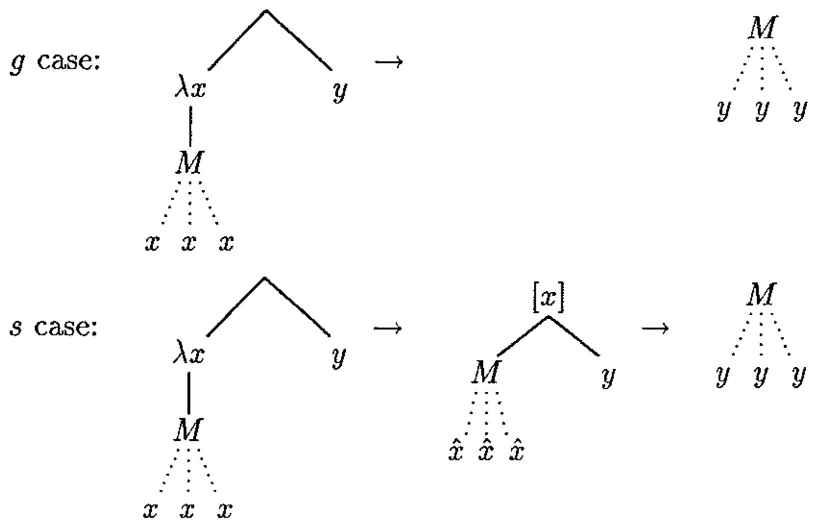


Figure 4.12: β-reduction followed by [triv-ptee] rule

$$\begin{array}{rlcccc} g \text{ case:} & (\lambda x.\{x_1\}\ N) & \rightarrow & N^\dagger & \equiv & N \\ s \text{ case:} & (\lambda x.\{x_1\}\ N) & \rightarrow & [_x\hat{x}_1\ N] & \rightarrow & N \end{array}$$

$$\begin{array}{rlcccccc} g \text{ case:} & (\lambda x.\{x_1\}\ N^\dagger) & \rightarrow & N^\dagger & \equiv & & & N \\ s \text{ case:} & [_y(\lambda x.\{x_1\}\ \hat{y}_1)\ N] & \rightarrow & [_y[_x\hat{x}_1\ \hat{y}_2]\ N] & \rightarrow & [_y\hat{y}_1\ N] & \rightarrow & N \quad \square \end{array}$$

Lemma 4.5 *A β_g-reduction $T_g^{\text{ic}} \rightarrow T_g^{\text{ir}}$ in which the rand is a lone variable corresponds to a β_s-reduction $T_s^{\text{ic}} \rightarrow T_s^{\text{ir}}$ followed by an application of the* [triv-ptee] *rule.*

Proof. The λ_g-interpreter treats this case specially by substituting copies of the single-node rand (not sharing them) and *not* marking them as SFEs Figure 4.12 shows the graph-versus-suspension comparison; the x's sticking out of M represent whatever bound variables happen to be in there (at least one). Note the absence of daggers representing SFEs. □

4.5.3 Moving λ_s-abstractions above suspensions

Another problem with using suspensions and λ_s-pointers is that a λ_s-application may be arbitrarily far from its plain-node rator, even without trivial suspensions; Figure 4.13 illustrates the problem. (An analogous difficulty

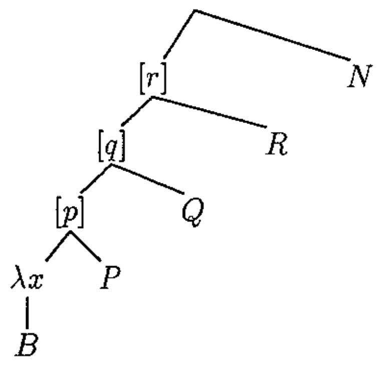


Figure 4.13: λx rator far from the plain-node λ_s-application above

for the λ_g-interpreter would be to have many indirection nodes between a λ_s-application and its rator.) As long as λ_s-terms of this form are allowed, β_s-reduction is not a local operation in the λ_s-tree.

The solution to the problem is to move a λ_s-abstraction suspension body above the suspension itself—the [λ-up] rule:

$$[\lambda\text{-up}]\text{: } [_x\lambda y.\{B\}\ P] \rightarrow \lambda y.\{[_xB'\ P^{\uparrow y}]\};$$

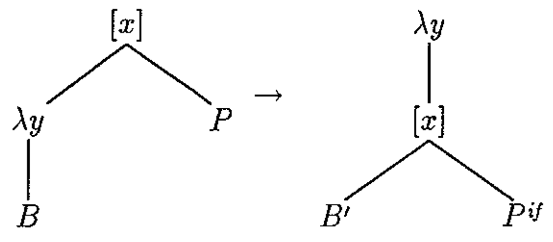
where binding indices must be adjusted as follows: free variables in P must be incremented, because of a new binder λy in their binding path, and binding indices of bound variables of λy and $[x]$ in B are incremented and decremented, respectively, so that they still point to the correct binder. Figure 4.14a shows the [λ-up] rule in λ_s-tree form.

The [λ-up] rule guarantees that a λ_s-application redex and its λ_s-abstraction rator will become adjacent in the λ_s-tree, even if there are intervening suspensions on the abstraction's root path. The only other non-locality between redex and rator that is possible is if a λ_s-pointer must be followed to get from one to the other; λ_s-pointer-marking and subsequent λ-filling take care of that.

The [λ-up] rule is simply one clause of the definition of substitution (see Hindley and Seldin [96, page 7]).

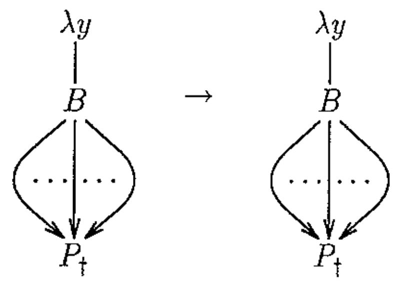
Lemma 4.6 *The* [λ-up] *rule preserves* λ_{sg}*-equivalence.*

Proof. The λ_g-graph equivalents (use Algorithm 4.1) of both sides of the rule are the same, as Figure 4.14b shows. The dagger indicates that P is a SFE, and the set of arrow-tipped lines suggest one or more pointers to P. □



$$[_x\lambda y.\{B\}\ P] \quad \to \quad \lambda y.\{[_x B'\ P^{if}]\}$$

(a) The [λ-up] rule (s case)



(a) Graph-reduction equivalent (g case)

Figure 4.14: The [λ-up] rule and its graph-reduction equivalent

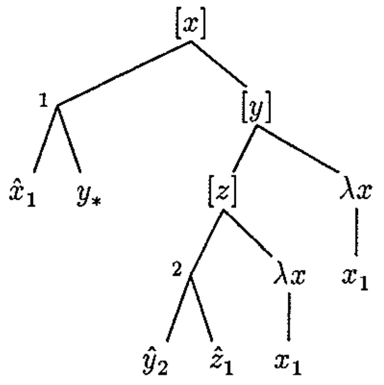
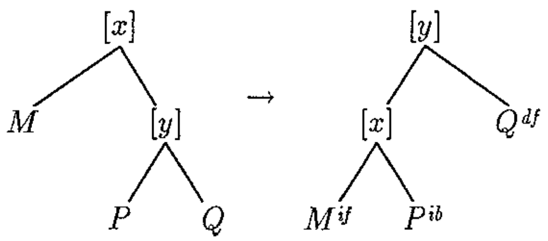


Figure 4.15: Suspension reordering needed



Free variables in M are incremented; variables in P bound to $[y]$ are incremented; free variables in Q are decremented.

Figure 4.16: Rotate adjacent suspensions leftward

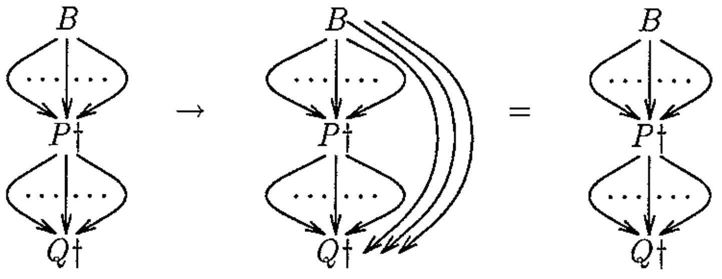
4.5.4 Rotating suspensions

Even without trivial suspensions, getting from one plain node to the next can still be messy; Figure 4.15 shows an example in which the desired hop is from λ_s-application 1 to λ_s-application 2: a λ_s-pointer hop up to suspension $[x]$, then a downward plunge through several suspension bodies (in general, there could be an arbitrary number). The problem is that suspension $[x]$ has another suspension, $[y]$, as its pointee. Put another way, one s-connection simply connects to another one.

The [sus-rotl] rule solves this new problem (it is called the "[sus-rotl]" rule because the suspensions rotate to the left):

$$\text{[sus-rotl]: } [_x M\ [_y P\ Q]] \rightarrow [_y [_x M^{if}\ P^{ib}]\ Q^{df}]$$

Figure 4.16 shows the [sus-rotl] rule in tree form. The effect of the rule is to turn suspensions unseparated by plain nodes into long left-linear suspension trees. (The "suspension-list" extension builds upon this property; see Section 6.1.)

g case: 

s case: $[_x B\ [_y P\ Q]] \quad \rightarrow \qquad\qquad [_y[_x B\ P]\ Q]$

Figure 4.17: The [sus-rotl] rule preserves λ_{sg}-equivalence

The [sus-rotl] rule follows from lemmas about substitution in Hindley and Seldin [96]:

$$\begin{array}{rcll} [_y M\ [_x P\ Q]] & \equiv & [_y[_x M^{if}\ Q]\ [_x P\ Q]] & \text{introduce a useless suspension;} \\ & & & \text{Lemma 1.14b.} \\ & \equiv & [_y[_x M^{if}\ P^{ib}]\ Q^{df}] & \text{Lemma 1.15d.} \end{array}$$

Lemma 4.7 *The* [sus-rotl] *rule preserves* λ_{sg}*-equivalence.*

Proof. The λ_g-graph equivalents (use Algorithm 4.1) of both sides of the rule are the same, as Figure 4.17 shows. The daggers indicate SFEs, and the set of arrow-tipped lines suggest one or more pointers. The possibility of pointers from B into Q is eliminated because the initial position of the $[y]$ suspension is invisible to any λ_s-pointers in B. □

4.5.5 Upward and downward s-connections

Recall that a g-connection is an edge that directly connects two plain nodes in a λ_s-term (or a λ_g-graph).

Plain nodes connected via a λ_s-pointer are *s-connected*. Figure 4.18 shows the only two ways a pair of plain nodes can be s-connected in a tidied λ_s-term. Figure 4.18a shows an *upward* s-connection: a plain node a has a λ_s-pointer child that s-connects it to the pointee b of the pointer's target suspension. The dashed line shows the s-connection. Tidying guarantees that a is not a suspension (else [triv-body]) and that b is not a suspension (else [sus-rotl]) or a variable (else [triv-ptee]).

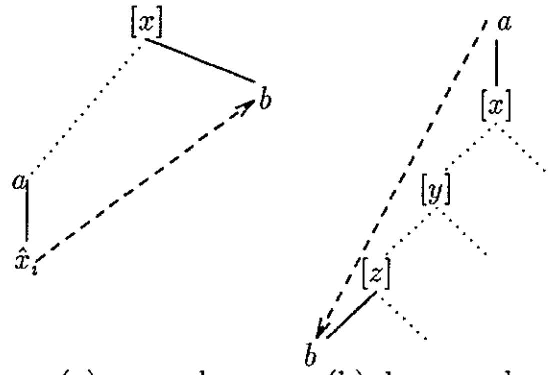


Figure 4.18: Kinds of s-connections

Figure 4.18b shows a *downward* s-connection: two plain nodes a and b have one or more suspensions hung between them. The dashed line is the s-connection. If there were another plain node c somewhere in the left-linear tree of suspensions, the a would be s-connected to c. The [triv-body] rule ensures that b is a plain node.

4.5.6 Constraints on moving suspensions

A suspension is a peg on a λ_s-tree on which a λ_s-term is hung so its bound variables may share it. By the nature of binding indices, a variable (here, I mean λ_s-pointers, too) can only "see" suspensions that are on the path from itself to the root of the λ_s-tree. A suspension may not be moved where one of the variables in either its body or pointee will no longer be able to see its binder.

As an example, consider Figure 4.19. The suspension $[x]$ could be moved to anywhere on the dotted lines. It cannot be moved further down, or one of its bound variables could not see it; it cannot be moved higher up, or the z's in its pointee could not see their binder.

All tidying rules observe these constraints.

4.5.7 Tidying: definition and important properties

Definition. A λ_s-term is *tidy* if none of the [useless], [triv-body], [triv-ptee], [λ-up], or [sus-rotl] rules applies to it.

A more informative definition of a tidied λ_s-term is possible:

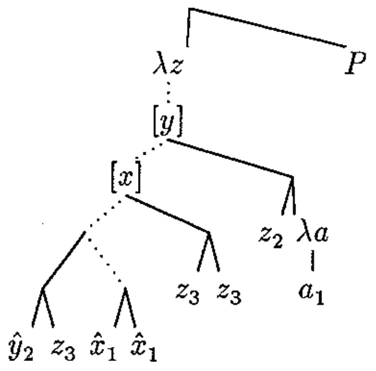


Figure 4.19: How far can suspension $[x]$ be moved?

1. The root (λ_s-application) and rator (λ_s-abstraction) of a redex are always directly connected (a g-connection).

   This guarantees that β_s-reduction is a *local* tree operation.

2. The λ_s-term includes no "garbage," nodes that are neither g- or s-connected to the term (i.e., nodes in useless suspensions).

   Aside from the impracticality of letting garbage accumulate, the FFPM implementation of searching for redexes (notably Algorithm 5.5) may not work correctly if useless suspensions are present.

3. The λ_s-term is organized so a traversal from one *plain* node to another crosses *one* connection, either a direct connection (g-connection) or an upward or downward s-connection.

   This constraint is essential to achieving comparable time complexities for the λ_g- and λ_s-interpreters.

The function tidyterm : Term $\rightarrow$ Term (page 88) encodes all the rules except the [useless] one; it is implemented by trashpickup (page 79). Table 4.3 shows all the λ_s-interpreter rules collected together, including those for tidying.

Important properties of tidying. β_s-reduction may "untidy" a λ_s-term, and the tidying part of a reduction step will tidy it up. The other parts of a step (searching and copying) must preserve tidiness. Searching for the next redex preserves tidiness trivially—searching does not change terms. The following lemma deals with tidiness preservation by lazy copying.

```
(* tidyterm : Term → Term.
   Innermost-out "tidying" of Terms with suspensions. Implements rules [triv-body],
   [triv-ptee], [λ-up] and [sus-rotl].
   Uses std_subst (page 172), incr_bd_vars (page 171), incr_free_vars2 (page 171), and
   swap_levs (page 173).
*)
fun tidyterm (App(M, N))    = App(tidyterm M, tidyterm N)
  | tidyterm (Lam(B, n))    = Lam(tidyterm B, n)
  | tidyterm (Var(x,vmk,n)) = Var(x,vmk,n)
  | tidyterm (Sus(B, P, n)) =
    let val B' = tidyterm B
        val P' = tidyterm P
    in (case B' (* we've tidied below; see what we got *)
            of Lam(lB, ln) => (* [λ-up] rule; may be more than one *)
               tidyterm (Lam(Sus(swap_levs 1 2 lB, incr_free_vars2 1 P', n),ln))
             | Var(bi, vmk,vn) => (* [triv-body] rule *)
               if bi ≤ 0 then (* a constant *)
                   Var(bi,vmk,vn)
               else if bi > 1 then (* keep body; heave away suspension *)
                   Var((bi-1),vmk,vn)
               else (* if bi = 1 *) (* replace body with suspension *)
                   (incr_free_vars2 ~1 P')
             | _ => (* otherwise, look at pointee *)
               (case P'
                  of Var(bi,vmk,vn) => (* [triv-ptee] rule *)
                     (incr_free_vars2 ~1 (std_subst P' B'))
                   | Sus(pB, pP, pn) => (* [sus-rotl] rule *)
                     let val B'' = (incr_free_vars2 1 B')
                         val pB' = (incr_bd_vars 1 pB)
                         val pP' = (incr_free_vars2 ~1 pP)
                     in
                        Sus(Sus(B'', pB', n), pP', pn)
                     end
                   | _ => (* otherotherwise... *)
                     Sus(B', P', n)
    )) end
```

β_s:	$(\lambda x.\{B\}\ P) \rightarrow [_x B\ P^{if}]$	
[useless]:	$[B\ P] \rightarrow B^{df}$	B contains no bound variables.
[triv-body]:	$[x_1\ N] \rightarrow N^{df}$	Cases in which the single-variable body is not bound to the suspension are covered by the [useless] pseudo-rule.
[triv-ptee]:	$[M\ y_i] \rightarrow M'^{df}$	M' is M with bound variables set to i; then decrement all free variables (including y_i).
[λ-up]:	$[_x \lambda y.\{B\}\ P] \rightarrow \lambda y.\{[_x B'\ P^{if}]\}$	Free variables in P are incremented; B' is B with bound variables of λy incremented and bound variables of $[x]$ decremented.
[sus-rotl]:	$[_x M\ [_y P\ Q]] \rightarrow [_y [_x M^{if}\ P^{ib}]\ Q^{df}]$	Free variables in M are incremented; variables in P bound to $[y]$ are incremented; free variables in Q are decremented.

Table 4.3: λ_s-interpreter rule summary

Lemma 4.8 *The λ_s-term that results from lazy copying into a* FollowFill *λ_s-pointer in a tidied λ_s-term (Section 4.4) is also tidy.*

Proof. Only g-connected plain nodes and λ_s-pointers are copied in as the rator of a λ_s-application; no suspensions are copied. None of the tidying rules can be applicable; they all involve suspensions. □

There are other important properties of tidying, besides those mentioned in the definitions above.

- No rule copies a λ_s-term bigger than a variable (one node); the [triv-body] and [triv-ptee] rules do this kind of "copying." All copying of larger terms is done when copying shared rators (Section 4.3).

- No rule requires a change in the left-to-right order of its subterms when written out in textual form. This is important for the implementation in Chapter 5, which would have to copy to change the order, but other implementations might suffer no such penalty.

4.5.8 The recurring example on the λ_s-interpreter

And finally, Figure 4.20 shows all the steps of the recurring example. You may wish to compare with Figures 2.8 (ordinary tree reduction) and 3.7 (graph reduction).

4.6 α-equivalence of λ_s-terms

λ-terms that are the same up to variable-renaming are said to be α-equivalent. In a name-free (suspensionless) λ-calculus, α-equivalent terms are *identical*. In an application where it is important to be able to determine equality of λ-terms, a name-free λ-calculus is used mainly for this reason. An example is Nadathur and Jayaraman's work on λ-Prolog [155].

Unfortunately, λ_s-terms (with suspensions) are not necessarily identical if equivalent, as Figure 4.21 shows. The (suspensionless) plain λ-term equivalents of both terms *are* identical, because the substitutions represented by suspensions may be done in either order (Hindley and Seldin's Lemma 1.15d [96, page 8]).

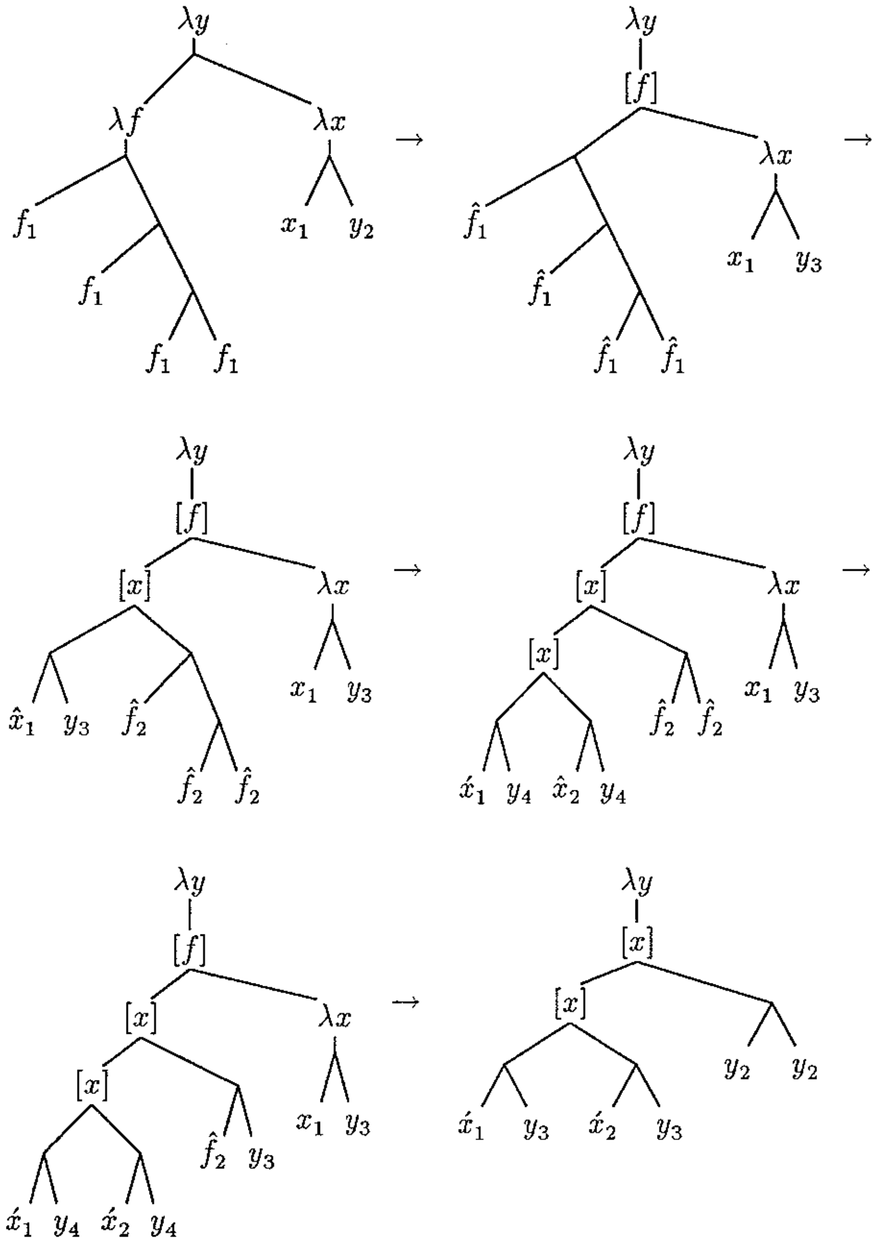


Figure 4.20: The λ_s-interpreter on the recurring example

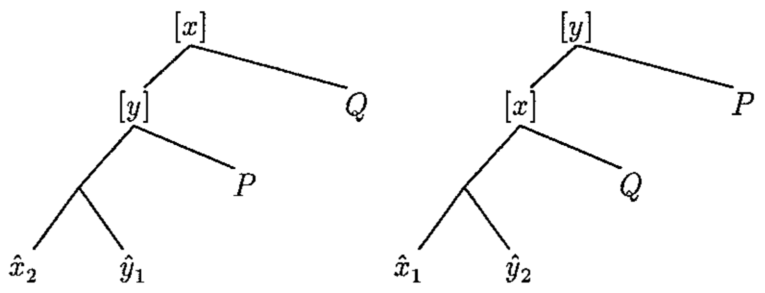


Figure 4.21: Equivalent, non-identical λ_s-terms

4.7 Equivalence to graph reduction: correctness

The guiding principle of a λ_s-interpreter is that it "does the same thing" as a λ_g-interpreter. The previous sections have showed that a λ_s-interpreter's actions are equivalent to a λ_g-interpreter in all phases of a reduction step: searching for a redex, copying the shared rator (if applicable), doing the β-reduction, and tidying up the result. Because Wadsworth showed that graph reduction is a correct implementation of the normal-order evaluation of the pure λ-calculus [201], it follows that the λ_s-interpreter is as well.

As the battle plans of Figures 4.1 and 4.2 proclaimed, the following theorem brings together the results about the correctness of the λ_s-interpreter that have been presented in this chapter.

Theorem 4.9 *Given an initial plain λ-term T_t^0 on which $n \geq 0$ normal-order reduction steps can be done and*

- *T_s^n, the result of doing n ($n \geq 1$) reduction steps on T_t^0 with a λ_s-interpreter,*
- *T_g^n, the result of doing n reduction steps on T_t^0 with a λ_g-interpreter,*

then $T_s^n \underset{sg}{\equiv} T_g^n$.

Proof. The proof is by induction on n. For the basis and induction steps, we must consider each of the searching, copying, reduction, and tidying phases. As before, the notation T_f^{ix} is to suggest a term in form f at reduction step i; x may be one of the letters s, c, r, or t, indicating one of the four phases of a reduction step.

Basis: searching. The input T_t^0 is a plain λ-term, an acceptable input for both the λ_g-interpreter and the λ_s-interpreter. All nodes are plain and all connections are g-connections. Both the λ_g-interpreter (which follows g-connections) and the λ_s-interpreter (which follows both g- and s-connections) will do a pre-order walk to the (same) first λ-application with a λ-abstraction for a rator.

Basis: copying. T_t^0 has no sharing because no reductions have happened yet, so the rator cannot be shared. Neither interpreter will do anything.

Basis: β-reduction. Lemma 4.1 says that λ_{sg}-equivalence is preserved for non-trivial, non-useless reductions. Lemmas 4.3, 4.4, and 4.5 prove the same thing for the useless-, trivial-body- and trivial-rand-reduction special cases.

Basis: tidying. If T_s^{0r} and T_g^{0r} are the results of the basis-step β-reduction and T_s^{0t} is the result of tidying T_s^{0r}, then $T_g^{0r} \underset{sg}{\equiv} T_s^{0r}$. Lemmas 4.3, 4.4, 4.5, 4.6, and 4.7 ensure that the [useless], [triv-body], [triv-ptee], [λ-up], and [sus-rotl] rules do not affect λ_{sg}-equivalence, respectively.

Thus ends the basis step for Theorem 4.9.

Induction step. The induction hypothesis is that the λ_g- and λ_s-interpreters each run for i steps, producing T_g^i and T_s^i, respectively, and that $T_g^i \underset{sg}{\equiv} T_s^i$. The goal is to prove that $T_g^{i+1} \underset{sg}{\equiv} T_s^{i+1}$.

Induction: searching. For every g-connection between plain nodes that the λ_g-interpreter follows, the λ_s-interpreter will follow an sg-connection between the corresponding plain nodes in the λ_s-term.

The λ_g-search will stop when its finds a λ_g-application g-connected to a λ_g-abstraction rator. The λ_s-interpreter looks for the same pattern; tidying guarantees a g-connection between redex and rator. The corresponding application nodes will be the redex.

In the λ_s-interpreter case, if the redex is connected to the rator by an s-connection, it will be an upward one. (A λ_s-abstraction cannot be the target of a downward s-connection, because of the [λ-up] rule.) Either a shared-rator copy or a last-instance relocation will follow. The latter cannot affect

λ_{sg}-equivalence, as its effect is only to replace the upward s-connection with a g-connection (Section 4.4).

Induction: copying. If the rator of the selected redex is not shared, both interpreters proceed to β-reduction. If the rator is shared, Lemma 4.2 says that the two interpreters do λ_{sg}-equivalent lazy copying.

Lemma 4.8 says that lazy copying of a shared rator in a tidied λ_s-term produces a tidied term.

Induction: β-reduction. As for the basis step.

Induction: tidying. As for the basis step. Thus ends the proof of Theorem 4.9. □

4.8 Related approaches to λ-calculus evaluation

This section reviews previous work about normal-order evaluation of the λ-calculus that is similar in some way to my approach in the λ_s-interpreter. A *vast* amount of work has been done on λ-calculus evaluation, much of it on practical variants, e.g., "reduction to weak head-normal form (WHNF) of an extended λ-calculus, using supercombinators." I do not trace the connections to that work, but limit myself to efforts closer in spirit.

Sections 3.5 and 5.1.12 review graph-reduction and non-graph-reduction architectures, respectively.

4.8.1 Efforts to find simpler reduction rules

I began this work in 1986 when Magó directed my attention to Staples's [190] and Révész's [174; 176] work on simpler sets of reduction rules for the λ-calculus. We examined their rules to see how they would fit on an FFPM.

In his work in the late 1970s, Staples's major concern was with "optimality theory," including finding an optimal reduction order for the λ-calculus (a provably-minimal number of steps to reach BNF) [189; 188; 190; 192; 193; 194] (Kennaway provides a summary of his work [118]). In the paper "A Graph-like Lambda Calculus for which Leftmost-Outermost Evaluation is Optimal" [190], Staples presents reduction rules for system that is equivalent

β_s:	$(\lambda x.\{B\}\ N) \to [_x B\ N^{if}]$	
[triv-body]:	$[_x x_i\ N] \to x_i$	if $i \leq 0$ (x constant)
	$\to N^{df}$	if $i = 1$ (pointer bound here; replace with pointee)
	$\to x_{i-1}$	if $i > 1$ (pointer bound above; pointee is unpointed to)
$\sigma\lambda 0$:	$[_x \lambda y.\{B\}\ N] \to \lambda y.\{B^{df}\}$	Only if there are no suspension bound variables (i.e., useless suspension); free variables in B must be decremented.
[λ-up]:	$[_x \lambda y.\{B\}\ N] \to \lambda y.\{[_x B'\ N^{if}]\}$	Free variables in N are incremented; B' is B with bound variables of λy incremented and bound variables of $[_x]$ decremented.
$\sigma\alpha$:	$[_y(((x\ A_1)\ A_2)\cdots A_{n+1})\ P] \to ([_y(((x\ A_1)\ A_2)\cdots A_n)]\ [_y A_{n+1}\ P])$	

Table 4.4: Staples's "graph-like lambda calculus" rules

to the λ-calculus for reductions to BNF where they exist (but not for arbitrary reductions). Table 4.4 shows Staples's rules (converted to my notation and to name-freeness). I use my rule-names where applicable.

The $\sigma\alpha$ rule only applies if a term has a variable at its head position (cf. Head-normal form (HNF)); Staples's scheme may be made fully equivalent to the λ-calculus if rule $\sigma\alpha$ is replaced with:

$$\sigma\alpha' : \ [_y(A\ B)\ P] \to ([_y A\ P]\ [B\ P]).$$

The effect of the $\sigma\alpha$ or $\sigma\alpha'$ rules is to push a suspension down through a λ-application; this is more obvious in the tree form shown in Figure 4.22.

Staples's system works by using the β_s rule to make suspensions and the $\sigma\lambda 0$, [λ-up], and (one of the) $\sigma\alpha$ rules to push suspensions downward in a λ-term until the [triv-body] rule applies.

The $\sigma\lambda 0$ removes a useless suspension and serves little purpose in a name-free calculus. I have altered Staples's [λ-up] rule slightly to take advantage of the name-free calculus.

For my purposes, the objection to Staples's rules is that the $\sigma\alpha$ rules

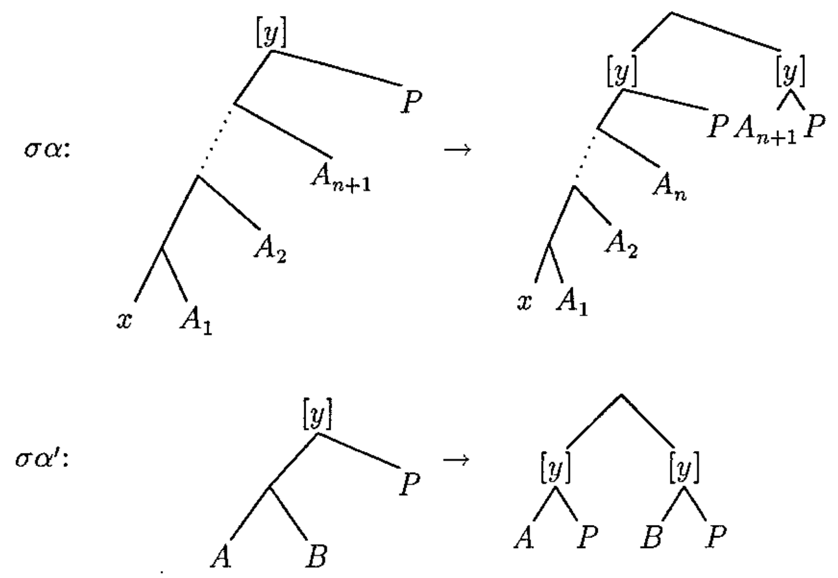


Figure 4.22: Staples's $\sigma\alpha$-rules

duplicate the suspension's pointee. It is to avoid this (possibly unnecessary) copying that the λ_s-interpreter follows λ_s-pointers and fills them in when necessary.

The [sus-rotl] rule in the λ_s-interpreter would serve no purpose in Staples's scheme, even though it is sensible (not wrong). More exhaustive checking for useless suspensions would be appropriate in Staples's system.

Michael O'Donnell [161, pages 59–62] follows Staples's remark about not using special symbols for substitution (i.e., suspensions) [190, page 441] and uses λ-abstractions for the purpose instead. O'Donnell also uses binding indices to give a name-free calculus; his resulting rules are essentially the same as Staples's. In related work, O'Donnell and Strandh worked on a similar system (interestingly, *with* an explicit symbol for substitution) in which they tried to avoid the adjustment of binding indices as reduction proceeds [162]. Their approach was to add an integer tag to every λ-term (every node, in λ-tree terms) and to augment their rules to manipulate the tags. They could not get it to work; O'Donnell decided that a single number cannot hold the information required (personal communication). Repeatedly adjusting binding indices is not a problem for the implementation of the λ_s-interpreter presented in Chapter 5.

Révész also presents a simpler set of reduction rules for the λ-calculus [174; 176]; as with O'Donnell, he is trying to break down substitution into

Rα:	$\lambda x.\{P\} \rightarrow \lambda z.\{[z//x]P\}$	where z is a 'fresh' variable
Rβ1:	$(\lambda x.\{x\}\ Q) \rightarrow Q$	
Rβ2:	$(\lambda x.\{P\}\ Q) \rightarrow P$	if x is not free in P
Rβ3:	$(\lambda x.\{\lambda y.\{P\}\}\ Q) \rightarrow \lambda z.\{(\lambda x.\{[z//y]P\}\ Q)\}$	if $y \neq x$ is free in P, z a 'fresh' variable
Rβ4:	$(\lambda x.\{(P_1\ P_2)\}Q) \rightarrow ((\lambda x.\{P_1\}Q)(\lambda x.\{P_2\}Q))$	if x is free in $(P_1\ P_2)$

Table 4.5: Révész's reduction rules

simpler steps. He uses "brute force" variable-renaming as part of his solution to the name-capture problem ("$[z//x]E$" means to rename x in E as z). Table 4.5 shows Révész's reduction rules (NB: *not* name-free variables).[3]

Révész's system is interesting, both as an elegant reformulation of the λ-calculus rules and as the basis for a practical implementation. Combining it with an extension to integrate lists into the λ-calculus [175; 176], Révész has built an interpreter for his language on the RP3 shared-memory multiprocessor [177].

The Staples and Révész systems are similar. Staples uses an explicit symbol for suspension while Révész does not. Their main difference is in variable naming; Révész uses his brute-force renaming, whereas Staples depends on an infinite supply of "fresh variables."

4.8.2 Comparison with environment-based evaluation

The problem of how to best implement β-reduction (in particular the core problem of substitution and the binding of variables) has received considerable attention; Kennaway and Sleep give a succinct synopsis of known approaches [121]. I have already covered *reduction* methods, in which program and data are both represented in a program graph (the graph is a tree for tree reduction), and the structure of the graph reflects binding information. Substitution is then a matter of copying, either pointers (graph reduction) or terms themselves (tree reduction).

[3]This is "axiom system A_2" from Révész's 1985 paper [174]; in his 1988 text, he concentrates on axiom system A_0, which does not include brute-force renaming [176].

Combinator-based approaches to reduction (orthogonal to whether tree or graph reduction is used) get rid of variables at compile-time, altering the requirements for an interpreter (Section 2.7 introduces combinators, and combinator-based architectures are included in the review in Section 3.5). Major flavors of combinator reduction include the use of a fixed set of combinators (as with Turner's combinators [199] or categorical combinators [50]), of program-specific sets of combinators ("super-combinators") [105], and of director strings [122]. Goldberg's paper on using abstract interpretation to detect sharing at compile-time is noteworthy with respect to supercombinators and sharing [81]. As I say in Section 2.7, combinators are far from my concern with the normal-order evaluation of the untransmogrified λ-calculus; I do discuss some overlaps later in this section, particularly with director strings.

The other main approach to implementing the λ-calculus is to use an *environment*: a set of variable-to-λ-term bindings, usually recorded in a separate data structure. Any evaluation of a λ-term takes place in an environment; bindings are added, perhaps modified, and looked up. Environment-based interpreters date back to LISP [149] and the SECD machine [130]; both are applicative-order evaluators, meaning that only fully-evaluated terms are stored in an environment.

Environments become more complex when they hold unevaluated terms, as in normal-order evaluation. Not just the λ-term, but also the "context" in which any later evaluation must take place (i.e., the bindings of free variables) must be recorded. Such a structure—a λ-term plus bindings for its free variables (an environment)—is often called a *closure* (e.g., Arvind et al. [9]).

Using environments allows different sharing properties, depending on whether an unevaluated λ-term is overwritten with its evaluated equivalent after that term has been reduced. Field and Harrison give a fascinating synopsis of what sharing can be achieved for various types of underlying implementation languages; for example, a fully-lazy-copying interpreter cannot be implemented with a fully-eager functional language [66, pages 208–211].

Some people use the term "suspension" to mean "an updatable environment," one that is modified in place, analogous to overwriting the root of a redex in graph reduction [127]. A suspension as I have defined it is in the same vein, except that only one variable is bound. My use of the term "suspension" comes from Staples [190]. He says "suspension is well-known in the theory of the classical lambda calculus" (page 441) and cites Rosen [178] and Mitschke [154] as antecedents. As I have mentioned, a "closure" is closely related; it is an environment plus a λ-abstraction, i.e., a function, hence the older name "FUNARG," from LISP. Yet another term comparable to "suspension" as I have used it is "recipe," cited in Field and Harrison's text [66, page 205]. In his thesis, Paul Watson uses "frozen substitutions" [206] and the term "delayed substitutions" also comes up. My equivalent of a more traditional environment—a *set* of bindings—is the set of suspensions along a root path. A λ-term plus the suspensions along its root path is my equivalent of a closure. Bound variables—their binding indices, that is—are indexes into the environment.

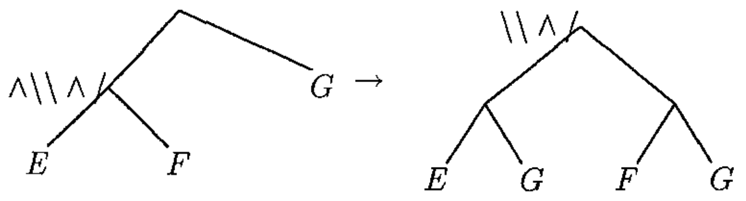
The categorical-combinators approach taken in the Categorical Abstract Machine also uses binding indices to index into an environment [51]. They use an explicit environment, whereas the λ_s-interpreter is really closer to reduction, with suspensions being an integral part of the program-plus-data structure to which reduction rules are applied.

As suggested earlier, *director strings* are related to traditional combinators; the motivation that Kennaway and Sleep give for them is similar to my suspensions:

> It would be better to only do the copying in response to the demands of the rest of the computation. One method of achieving this is to introduce environments... When we encounter a beta-redex $(\lambda x.\{F\}\ G)$, we merely replace it with the pair $[F, \langle x = G\rangle]$, where $\langle x = G\rangle$ is the environment that associates x with G... We then continue by attempting to evaluate F. If we discover further redexes, we reduce them. But if we find an occurrence of x whose value we need before proceeding further, then we 'push' the environment $\langle x = G\rangle$ down through F to that occurrence of x, peeling off a copy only of the path traversed. At the end of the path we substitute for x a pointer to G, and continue looking for the next redex to reduce [121, page 120].

Kennaway and Sleep call this approach "lazy graph reduction" and suggest director strings as one implementation. These are notations in a λ-tree that show how arguments should be pushed down the tree. The possible directors are /, \, ^, and -, meaning send "to the left," "to the right," "both ways," and "nowhere," respectively. Figure 4.23 shows a λ-application with five directors, as well as the result of applying one argument.

The approach taken in the λ_s-interpreter differs from director strings in the same way it differs from Staples's rules: I avoid the potentially useless copying of "sending both ways." Kennaway and Sleep suggest a graph-reduction base, so they need not fear such copying.



$\wedge\backslash\backslash\wedge/$
E
F
G →
$\backslash\backslash\wedge/$
E
G
F
G

Figure 4.23: An example of director strings

4.8.3 Environment/reduction hybrids

My suspensions are single-binding environments. The λ_s-interpreter is unusual in that it works both in a "reduction mode" (e.g., the rules that move suspensions around) and in an "environment mode" (when following and filling λ_s-pointers).

In their paper on sharing in functional language implementations, Arvind et al. also describe a hybrid reduction/environment scheme [9, pages 5.5–5.6]. It has "graph cells" that may point to "environment cells," as well as to each other. (The environments are of the traditional multi-binding kind.) Reduction at a (graph) cell takes place in the context of the environment cell attached there. Pointers to environment cells are passed down the graph as evaluation proceeds; evaluation inside an environment may also be required. Reduction in which environments (or closures, or suspensions) are stored and manipulated as part of the graph (or tree) are *closure-based.*

Fairbairn and Wray's Three Instruction Machine (TIM) is a well-developed example of closure-based (supercombinator) reduction, intended for use with stock hardware [64] (I follow the description by Koopman and Lee [127]). When a combinator evaluation is to be delayed, the current stack frame contains pointers to the ancestor nodes of the combinator—that is, the environment in which the evaluation will be done if necessary. To delay a reduction, TIM copies the current stack frame into a closure in the heap, in much the same way as registers are copied to memory on a context-switch. These closures have much the same flavor as "suspension lists," introduced in Section 6.1.

4.8.4 Pointers versus λ_s-pointers

A graph reducer like the λ_g-interpreter (Chapter 3) depends on the notion of pointer, a unique identifier for an entity in a global name space: it provides an *absolute* address of something. In the λ_s-interpreter, I use binding indices to point to suspensions' pointees; these λ_s-pointers are, in some sense, *relative* addresses, saying how far to walk up a λ-tree.

Using absolute pointers suffers when a λ-term is copied; all pointers with targets in the old copy must be changed to pointers into the new copy. On the other hand, a λ_s-term, with relative λ_s-pointers, can be copied bit-for-bit and all will be well.

Absolute pointers regain some merit in other operations, including substitution (of pointers): the absolute pointer can be put in for the bound variables, no questions asked. Relative λ_s-pointers, *au contraire*, must be repeatedly adjusted as binding depths change. In Chapter 5, however, I show that one can have a machine implementation in which the "disadvantages" of relative λ_s-pointers do not arise.

Chapter 5

The λ_s-interpreter on an FFP Machine

Representation *is* the essence of programming.
— Frederick P. Brooks, Jr. (1975).

This chapter describes how to implement the λ_s-interpreter on an FFP Machine (FFPM), a small-grain MIMD computer architecture that supports functional programming. (The λ_s-interpreter is described in Chapter 4.) Following an introduction to the FFPM in Section 5.1 (including a review of other non-graph-reduction architectures aimed at functional programming), this chapter catalogs the FFPM algorithms needed for a λ_s-interpreter. Section 5.2.1 gives algorithms for basic operations that are pervasive in an interpreter (e.g., detecting bound variables), and the following section sets out the rest of the implementation in a top-down fashion, with the highlights reviewed in Section 5.2.7. Section 5.4 goes over previous FFPM work related to the λ-calculus.

The algorithms in this chapter are presented in English, with examples; a more formal presentation would heighten the tedium to an unbearable level. I give time and space complexity results as I go along; I use the symbol "lg" for "binary logarithm."

I assume that an FFPM is not reducing several independent programs (λ_s-terms) at once; this slightly simplifies the overall control of the reduction process (and the description thereof).

5.1 Introduction to the FFP Machine

5.1.1 Project history and design goals

Gyula Magó began the FFPM project in the mid-1970's at the University of North Carolina at Chapel Hill; his goal was to design a highly parallel machine to support functional languages, John Backus's FFP language in particular [14]. Magó published the first description of an FFPM in 1979 [141], with follow-on descriptions by Magó and Middleton in 1984 [140], and Magó and Stanat in 1989 [147]. (The last is the best available description of the "official" FFPM; Almasi and Gottlieb's book includes a good brief description [4].) Besides Magó and Stanat, some fifteen graduate students have put their fingers in the pie over the years; I am one. Hundreds of FFPM variants have popped up, with half-lives as short as one cup of coffee.

Though some design choices have changed during the project, its goals have not varied much. Magó's paper "Making Parallel Computation Simple" [144] outlines these goals and design postulates:

- As much attention should be paid to ease of programming as to execution speed. An FFPM system manages all parallelism without programmer intervention (implicit parallelism).

- For the same reason, storage management (garbage collection) should be fully automatic, even built in hardware.

- The design should be scalable in small increments up to an indefinitely large size. The design is *cellular*, built from many copies of a few simple VLSI parts.

- A scalable design eschews shared system resources; consider a million processors attached to shared memory, for example. Locality becomes very important, and an FFPM arranges that an independent subcomputation always runs on physically-proximate hardware, entirely unaffected by the rest of the computer system.

- Similarly, sharing a planning resource, e.g., a master processor, is out of the question, so an FFPM has no central control.

- An FFPM is intended for dynamic computations, those with unpredictable data-structure sizes and shapes and with unpredictable threads of control. The hardware should be dynamically mapped onto the computations, rather than the computations painstakingly mapped onto hardware at program-design time.

- The scalability criterion excludes programs with special knowledge of the Machine: number of processors, interconnection topology, etc.

The FFPM has been examined from many angles, as one might expect of an unusual design. (The interested reader would do well to get the FFPM project bibliography [145].) Among these angles has been support for non-FFP languages (machine name notwithstanding). The earliest designs tracked Backus's still-developing ideas about reduction languages (compare the original 1979 paper [141] with the 1989 one [147]). Bruce Smith has studied logic programming on an FFPM [185; 186]. Dybvig described how to support the full Scheme language [62]. Middleton and Smith [153] explored the potential of the FFPM equivalent of microprogramming; they later applied some of their ideas in sketching an implementation of OPS5 production systems [187].

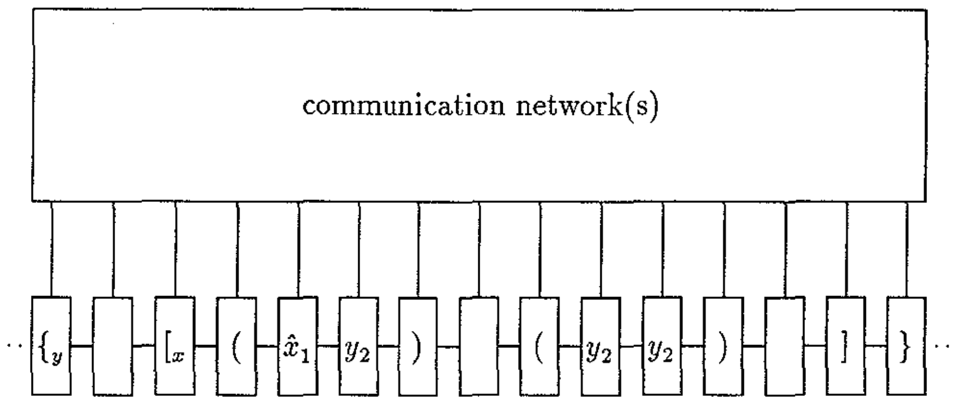
Despite the many directions of study and design-of-the-week graduate-student enthusiasms, a set of core features for an FFPM has emerged—things that come up in practically every design and that seem to be essential even when doing non-FFP things. All this is to say that my description of an FFPM does not exactly match any standard FFPM description, but it is close.

I defer the history of λ-calculus implementation on the FFPM until Section 5.4.

5.1.2 Basic structure of an FFP Machine

The heart of an FFPM is a linear array of small cells, each with its own CPU, memory, and communication hardware, capable of independent execution (that is, the Machine is MIMD). This string of cells is called the *L-array*. The cells do not have globally-known addresses, and actions take place because of their *contents*; the L-array is, in some sense, an associative memory.

Program symbols are laid out on the L-array in a straightforward, linear representation, one symbol per cell, preserving the left-to-right order of the symbols as they appear on paper. Figure 5.1 shows how the term $\lambda y.\{[_x(\hat{x}_1\ y_2)\ (y_2\ y_2)]\}$ might appear in an L-array. The explicit λy information need not be kept; instead, I show the (decorative) name as a subscript



L-array of processing cells

Figure 5.1: Structure of an FFP Machine

on the left brace, $\{_y$.[1] A machine implementation would certainly throw out variable names, keeping only binding indices. The figure also shows how empty cells may be scattered through a program; we will ignore them. *Storage management* in an FFPM shifts the program symbols along the L-array, preserving left-to-right order. Storage management must allow terms to grow and shrink. Addresses for the cells would serve little purpose because a cell's contents may change from cycle to cycle. Singh and Chi give a design for FFPM storage-management hardware [183].

The built-in storage management allows an FFPM to have arbitrarily nested *dynamic arrays* as its basic data structure [146]. A dynamic array allows insertion and deletion anywhere in the structure (as in LISP lists), while allowing constant-time access to its elements (as in FORTRAN arrays).

Cells in an FFPM compute by changing their symbols in orderly ways that correspond to the effects of reduction rules. In other words, each cell executes some small conventional-looking microprogram—called a *reduction routine*—that may change the cell's symbol. Obviously, some communication among cells is required, as is the insertion and deletion of symbols. In this dissertation, I ignore questions of where reduction routines come from, where they are kept, and how a cell knows which one to use (Danforth's description of his simulator covers these issues [55]).

An FFPM can do any kind of string reduction, rewriting one arbitrary

[1]Many program representations are possible in an FFPM, and this is a crucial question in real FFPM design; Middleton has done the main work [152].

string of symbols into another; a crazy-looking rule—for example, $arBst \rightarrow xyxytreB$, would be feasible. In this sense, an FFPM is a "string reduction" machine. In practice, however, the rewriting is always concerned with "strings" of a specific flavor: flattened representations of parse trees. All FFPM design has concentrated on improving this kind of reduction; therefore, it is more reasonable to call the Machine a *tree reduction* architecture.

5.1.3 Communication and partitioning

Reduction in an L-array will not get far unless the cells can talk to each other—every FFPM design will have an interconnection network perched atop the array, as in Figure 5.1. Traditionally, there is a set of binary tree networks augmented with limited computing capability. (The processing cells make up the *Leaves* of the network; hence the term "L-array.") Kellman [117] and Plaisted [170] have proposed richer networks for an FFPM; any networks that meet the requirements below would be appropriate. All analysis in this dissertation assumes the standard tree networks.

In addition to the main networks, the L-array cells have lateral connections between them for storage management. If an FFPM had a richer network, the lateral connections could be dropped.

The tree networks support *global* network operations in which all cells in a Machine participate. More interesting, however, is that the L-array of an FFPM can be partitioned into *L-segments*, and each segment can be allocated an independent sub-network all its own. Because they use different wires at the physical level, global network operations and those on L-segments' separate subnetworks proceed concurrently. (I am glossing over the details of wiring, switches, buffers, etc., in the networks; Magó and Stanat's review says more [147].)

Figure 5.2 shows partitioning with left and right parentheses as delimiters:[2] the *innermost, delimited* terms have subnetworks allocated to them (shown by solid lines). Partitioning is fast (one bit from each cell must travel to the root at hardware speeds), is unconstrained by program layout, and puts an isolated tree sub-network over each segment. Each segment/subnetwork combination proceeds independently; all needed resources are *local*. The nemesis of a tree network—a bottleneck at the root—is alleviated by the potential for having *many* tree subnetworks running at once.

[2]Reduction-routine code in the cells could choose other program symbols to delimit the segments.

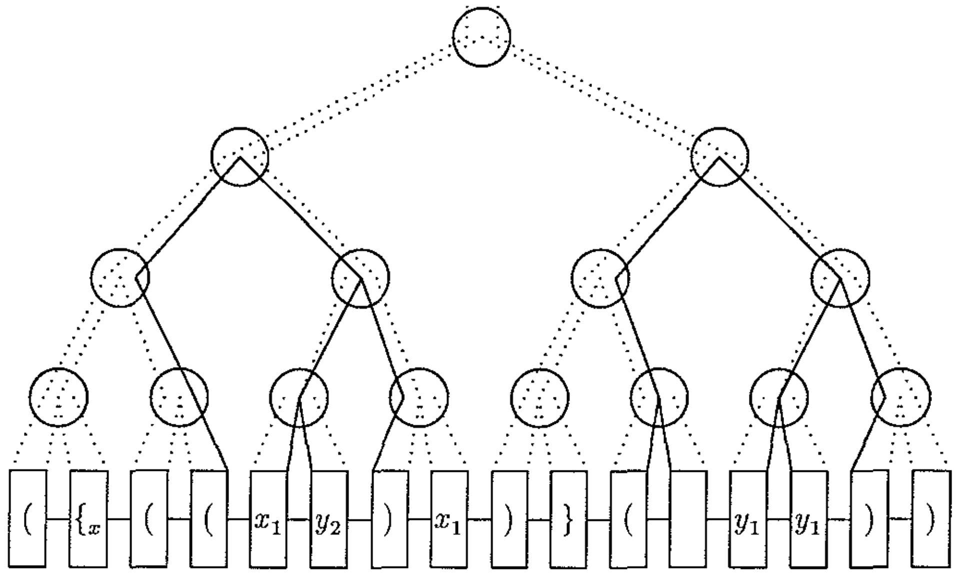


Figure 5.2: Partitioning in an FFP Machine

If an FFPM is partitioned with syntactic delimiters, e.g., parentheses denoting applications, only innermost delimited-strings will be self-contained in L-segments. These L-segments, the only ones with all information locally, are said to be *active*. It is these active L-segments that do reductions; therefore, at the hardware level an FFPM implements *applicative-order* reduction.

To override an FFPM's built-in innermost reduction, one chooses (and manipulates) partitioning delimiters more craftily. For example, to do outermost reductions first, one might use "marked" parentheses to delimit partitioning. To begin, only the outermost pair is marked, and the whole expression will be in the initial (active) L-segment. Code in that first L-segment may then choose to mark some inner parentheses. At the next partitioning, the newly-marked inner L-segments will become active. If those inner L-segments want to cede control back outward, they just unmark their parentheses. Middleton and Smith showed how such partitioning tricks can be used for clever non-FFP purposes (e.g., simulating systolic arrays) [153; 187]. They used parentheses as the partitioning delimiters, the marks on them were called "colors," so these partitioning tricks flew under the banner of "colored parentheses."

The other communication paths in an FFPM are at its extremities. First, the lateral connections of the endmost cells connect to some form of "virtual memory," which deals with expressions too large to fit in the L-array; Geoff

Frank et al. investigated it [69; 70]. Second, the topmost nodes of the tree networks connect to a Front-End machine (not shown in Figure 5.1). I assume that this device can decide when to initiate partitioning, what partitioning delimiters to use, when to interrupt the independent processing by L-segments, and when to start storage management. I also assume that the Front-End can do simple associative-memory-like queries (e.g., "Is there a cell with a constant in it?"), allowing a data-determined machine cycle. Historically, FFPM designs have had a passive Front-End, with a hard-wired partitioning/execution/storage-management cycle; this is entirely reasonable when implementing FFP. I am just using what the hardware design has allowed all along.

5.1.4 Broadcasting and sorting operations

The simplest type of communication that can take place in an L-segment is the *broadcast* of a datum from one cell to all cells. One cell sends something useful, all the others send some null value. Eventually, all cells receive the useful value. With a tree network, in the worst case, a one-datum broadcast requires a trip up and down the entire tree (a *message wave*); it is a $O(\lg A)$ operation, where A is the size of the "address space," the number of cells in the Machine.[3]

If each cell in a segment sends a value and each tree node merges the values that come to it, then a sorted list of values will emerge from the root. That sorted list can be broadcast to the cells below; they, in turn, can select values of interest. If n values are sent from the cells, sorting is an $O(n)$ operation, because each value must pass through the root before being sent down.

In doing a sort operation, each cell could send several values, the first being a key, the rest "baggage." If the keys are known to be sorted in advance, such a sort could serve to broadcast the contents of a string of cells; another string of cells might replace their program symbols with data from the sort. In this way, strings of symbols can be copied or moved around. But it takes time $O(n)$ in a tree, too expensive to use as a basic operation in an interpreter on a parallel machine.

If broadcasting and sorting are done as global operations over all cells, the Front-End machine can observe the values that reach the root of the tree.

[3]Bruce Smith has showed that the *expected* time for an n-cell L-segment to do this basic operation is $O(\lg n), n \leq A$. He assumes no particular alignment and that storage management keeps program symbols distributed uniformly across the FFPM [186].

5.1.5 Single-result operations

In an FFPM, the network hardware can operate on the values sent up from the cells; examples include adding the received values or selecting the maximum (or minimum) or the leftmost (or rightmost). This single result is then broadcast to all cells (and, if on the global network, passed to the Front-End). As with the broadcast of a single datum, this operation takes $O(\lg A)$ time.

A single-result network operation may subsume the simple broadcast: using addition, the interesting cell sends its interesting value, and all other cells send 0—the result is a broadcast.

5.1.6 Scan, or parallel prefix, operations

An FFPM's network hardware supports *scan operations*. A (left-to-right) scan $\odot$ operates on a sequence of values $(i_1, i_2, \ldots, i_n)$ to produce a sequence of result values $(o_1, o_2, \ldots, o_n)$

$$o_j = i_0 \bullet i_1 \bullet i_2 \cdots i_{j-2} \bullet i_{j-1};$$

where i_0 is the identity value for the operation $\bullet$.[4] For example, here is an integer-addition scan (0 is identity value for $+$):

input:	2	7	-4	1	-2
result:	0	2	9	5	6

The results are, reading left to right, the total-so-far at each position. An integer-minimum (left-to-right) scan on the same sequence (using $+\infty$ as the identity value!):

input:	2	7	-4	1	-2
result:	$+\infty$	2	2	-4	-4

A scan operation $\odot$ makes sense if the binary operation $\bullet$ is associative. Also, a scan operation can go right-to-left; these scans are less common.

It is easiest to think about a scan as computing a running total at each position (a serial operation); however, a scan can be computed in parallel in $O(\lg A)$ time; Meijer and Akl establish this for tree networks [151; 3], and Magó and Stanat give the FFPM details [147]. Part of the processing power in an FFPM network node is to achieve this speed. Other parallel machines

[4]A variant definition is: $o'_j = i_0 \bullet i_1 \bullet i_2 \cdots i_{j-1} \bullet i_j$. A cell wanting that value computes it with its own value and the value from the network: $o'_j = o_j \bullet i_j$.

have special support for scan operations, notably the NYU Ultracomputer, with its fetch-and-op network (1983) [61; 82; 129], and the Connection Machine (1985) [93]; the latter promulgated the term "scan." Scans are useful; Blelloch goes so far as to call them a "model of computation" [35].

Scans are even more useful if they can be restarted, i.e., a cell can indicate "Restart the running total here." If restart values are marked by an "R" prefix, then an integer-addition scan would be (0 as identity):

input:	2	7	R-4	1	-2	R3	3	2
result:	0	2	9	-4	-3	-5	3	6

A scan with *every* value restarting is a right shift:[5]

input:	R2	R3	R4	R3	R5
result:	0	2	3	4	3

A comparable right-to-left scan is a left shift. Scans-with-restarting are still computed in parallel in $O(\lg A)$ time.

Scan operations are also called "parallel prefix" [128] or "cumulative sum" [140] operations. I use "scan" because it is shortest.

5.1.7 Computing level numbers

An FFPM reduction-routine setting out on its mission might need, e.g., to distinguish rator-cells from rand-cells in a term. To this end, the Front-End ensures that various *level numbers* are periodically (re-)calculated, with the results kept in the cells. These calculations are scans, and they are a good introduction to the low-level goings-on in an FFPM.

The simplest level numbering is a left-to-right count (1, 2, 3,...) of all (non-empty) cells, giving each cell its *index*. The index is a unique identifier for a cell—in effect, a temporary address until the symbol-string changes. It could be used to determine relative position: a special cell broadcasts its index; other cells then check if they are before or after the special cell. Indices must be recalculated after insertions or deletions of program symbols.

The index is calculated by each cell submitting a one to an integer-add scan; the left-to-right running total will be the index. Each cell increments its value, so the index will begin with one. Here is a complete example:

[5]An FFPM really does a *rotate* right (the rightmost value ends up at the left end), thanks to the particulars of its networks' design.

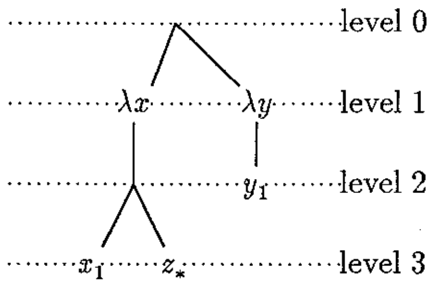


Figure 5.3: Nesting levels for a λ_s-term

	(	$\{_x$	(	x_1	z_*	)	}	$\{_y$	y_1	}	)
scan input:	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
scan result:	0	1	2	3	4	5	6	7	8	9	10
index:	1	2	3	4	5	6	7	8	9	10	11

The next level numbering is the *nesting level*; Figure 5.3 shows nesting levels for the sample λ_s-term. The heart of the calculation is a count of unmatched left syntactic-delimiters preceding a cell. It is an integer-addition scan, each cell contributing:

- 1, if it is a left syntactic-delimiter (, {, [,
- -1, if it is a right syntactic-delimiter), },], or
- 0, otherwise.

After the scan, right syntactic-markers' values are decremented so that they have the same nesting level as their corresponding left syntactic-delimiter. Here are the scan input, result, and (after adjustment) nesting level (compare to Figure 5.3):

	(	$\{_x$	(	x_1	z_*	)	}	$\{_y$	y_1	}	)
scan input:	+1	+1	+1	0	0	-1	-1	+1	0	-1	-1
scan result:	0	1	2	3	3	3	2	1	2	2	1
nesting level:	0	1	2	3	3	2	1	1	2	1	0

If a level number is calculated for the whole L-array, it is an *absolute* level number; e.g., if *all* the cells are numbered left-to-right, that is an *absolute index*. On the other hand, if a level number is calculated separately for each L-segment (by restarting the count at each segment left-delimiter or by doing the scans after partitioning) then it is a *relative* level number.

		Cell's contribution						
Abbrev.	**Level number**	(	)	{	}	[	]	other
AIX,RIX	index (absolute, relative)	1	1	1	1	1	1	1
ANL,RNL	nesting level	1	-1	1	-1	1	-1	0
ABL,RBL	binding level	0	0	1	-1	1	-1	0
AAL,RAL	application level	1	-1	0	0	1	-1	0

		$\vdash$													$\dashv$	$\vdash$									$\dashv$	
	(	(	$[_a$	a_1	z_*	]	$[_b$	$\{_c$	c_1	}	(	y_*	y_*	)	]	)	(	$\{_d$	d_1	}	$\{_e$	$\{_f$	e_1	}	}	)	)
AIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
RIX:		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	1	2	3	4	5	6	7	8	9	10	
ANL:	0	1	2	3	3	2	2	3	4	3	3	4	4	3	2	1	1	2	3	2	2	3	4	3	2	1	0
RNL:		0	1	2	2	1	1	2	3	2	2	3	3	2	1	0	0	1	2	1	1	2	3	2	1	0	
ABL:				1	1		0	1	2	1	1	1	1	1	0				1		0	1	2	1	0		
RBL:				1	1		0	1	2	1	1	1	1	1	0				1		0	1	2	1	0		
AAL:	0	1	2	3	3	2	2	3	3	3	3	4	4	3	2	1	1	2	2	2	2	2	2	2	2	1	0
RAL:		0	1	2	2	1	1	2	2	2	2	3	3	2	1	0	0	1	1	1	1	1	1	1	1	0	

Table 5.1: Calculating the standard level numbers

5.1.8 Calculating exotic level numbers

Other level numbers besides indices and nesting levels come from counting various sets of delimiters. If one only counts brackets (representing suspensions) and braces (representing λ-abstractions), then one is counting "binding depths" from the root of the term downwards. These are *binding levels.*

If one instead counts only parentheses (λ-applications) and brackets (suspensions)—the symbols that are outermost in all rules, the resulting numbers are *application levels.*

As before, absolute numbers are those calculated for *all* symbols in an FFPM, regardless of partitioning, and relative numbers are those calculated independently in each active L-segment. Since most action takes place in L-segments, relative numbers dominate.

I usually refer to nesting-level numbers by their acronym; Table 5.1 gives those I use, and the cells' contributions to the scan operations. The table also shows an example with all level numbers calculated; the markers $\vdash$ and $\dashv$ set off two active L-segments, each with its own relative level numbers; zeros are not always shown.

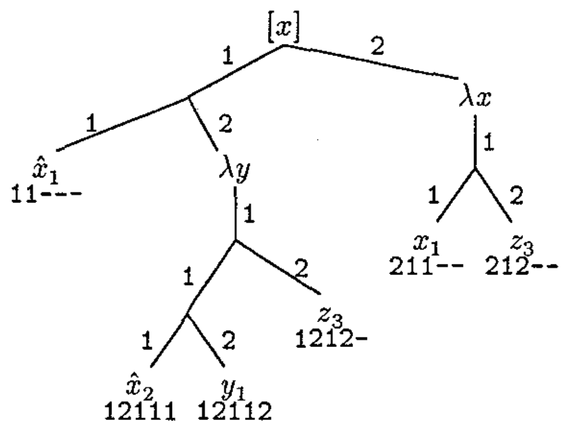


Figure 5.4: λ_s-term with its selectors shown

5.1.9 Calculating selectors and first/last bits

For a cell to participate in a reduction routine that implements a λ_s-interpreter rule, it must know "where its symbol is" in the λ_s-term. Sometimes, one of the level numbers is enough; for example, if the question is, "Am I on the same level as a particular cell?"

Sometimes a cell needs more information than a level number can provide; for example, "Am I in the rator or not?" *Selectors* provide the needed, limited parse-tree information. Consider Figure 5.4: the edges coming out of each node are numbered, `1,2,3`... The sequence of the edge-numbers from the root down to the node in question *selects* a node (subtree). I show a five-level selector for each variable in the figure.

To select deeply-nested subterms, we would need arbitrarily long sequences of edge-numbers. Fortunately, for the algorithms in this dissertation, *two* levels of selectors are enough. Only the algorithm for the [sus-rotl] rule (Algorithm 5.13, page 135) needs the second one.

Cells often need to know if they are the first or last cell in a term at a given nesting level (RNL) n. (Level 1 is the top level; in Figure 5.4, it chooses between the body and pointee of the top suspension.) A variable with RNL $= n$ is both First and Last at level n—a variable is a whole term. For delimiter cells with RNL$= n$, a left-delimiter is the First cell of a subterm on level n and a right-delimiter is a Last cell. Comparing the following diagram with Figure 5.4 should clear the air. (Zero entries shown as blank.)

	$[_x$	(	x_1	$\{_y$	(	(	x_2	y_1	)	z_3	)	}	)	$\{_x$	(	x_1	z_3	)	}	]
RNL:	0	1	2	2	3	4	5	5	4	4	3	2	1	1	2	3	3	2	1	0
First_1:		1												1						
Last_1:													1						1	
Sel_1:	0	⟨1	1	1	1	1	1	1	1	1	1	1	1⟩	⟨2	2	2	2	2	2⟩	0
First_2:			1	1											1					
Last_2:			1									1						1		
input:			+1	+1								R0	R0		+1			R0	R0	
result:			0	1	2	2	2	2	2	2	2	2	0	0	0	1	1	1	0	0
Sel_2:			⟨1⟩	⟨2	2	2	2	2	2	2	2	2⟩			⟨1	1	1	1⟩		
First_3:					1											1	1			
Last_3:											1					1	1			
input:			R0		+1						R0	R0				+1	+1	R0		
result:				0	0	1	1	1	1	1	1	0	0			0	1	2	0	
Sel_3:					⟨1	1	1	1	1	1	1⟩					⟨1⟩	⟨2⟩			

First_i and Last_i bits follow from RNL information; besides listing them explicitly in the example, I also show First_i and Last_i bits as ⟨ and ⟩ symbols on the Sel_i numbers, respectively. I will use the $<, >$ notation in the rest of the dissertation.

Selectors are calculated from the First_i and Last_i information. The top-level selector, Sel_1, is an integer-addition scan with cells sending First_1. Then, for level $n > 1$, an integer-addition scan (order is significant):

- Cells with First_n set send +1.
- Cells with Last_n or Last_{n-1} set send Restart 0. This is the all-important restart of the left-to-right count.
- Other cells send +0.

I show the inputs and results of the Sel_2 and Sel_3 scans in the example above. Cells add their own input to the scan result to get the selector.

5.1.10 Low-level programming style in an FFP Machine

The implementer of a language system on an FFPM writes reduction routines to manipulate cells' contents in accordance with the language definition. Scans and the communication facilities just outlined are the stock in trade. These routines often follow one of a few strategies; I sketch one *example* here.

The first thing a reduction routine does is a few scan operations to calculate various level numbers that a cell can use to determine its position in the term. In this way, a cell learns about itself: "Do I hold the last parenthesis?", "Am I in the rator or the rand?" or "Am I the second cell in a λ-application?" Answering these questions, a cell decides its action for this cycle. In an active segment, each cell will typically be doing one of a few actions. For example, all the cells in the rator would do one thing, and all those in the rand another; frequently, the parentheses delimiting the term do something different (such as deleting themselves). Once cells have determined their task, reduction will likely unfold in two phases. First, the cells will communicate to decide what space is needed for the impending rewriting of symbols; storage management will then provide the needed free cells in the requested slots. Next, symbols to be moved will be broadcast—or sorted if a reordering is taking place—and the free cells will swallow the incoming symbols. There are exceptions to this pattern, but it fits many cases .

Summarizing, an FFPM has a linear array of processing cells that cooperate to execute a reduction routine, thereby causing an efficacious rewriting of program symbols. The cells work in L-segments. Besides computing individually, the cells in a segment have their own sub-network that provides broadcast, sorting, single-result, and scan-operation services. A Front-End machine directs the overall computation. Storage management is automatic.

5.1.11 Copying and sharing in an FFP Machine

An FFPM tries to achieve high speeds by having many non-interfering computations proceeding at once. Each computation is self-contained, having all necessary program and data.

The FFPM philosophy does not consider space sharing (to conserve memory) an absolute good; sharing can be inimical to high performance. To get highly parallel computing in a scalable architecture, you *must* have multiple copies of program structures. In other words, an FFPM favors intelligent, even speculative copying.

There are some common programming idioms in which the pro-copying approach suffers badly; Magó addresses these concerns in his 1981 paper, "Copying Operands versus Copying Results" [142]. A typical expression that he seeks to optimize is

$$(\lambda x.\{\text{if } \textit{is-null}\ x \text{ then } x \text{ else } \textit{transpose}\ x\}\ \textit{huge-vector}).$$

To do the trivial *is-null* x test requires copying the whole *huge-vector* in for x. Magó shows how low-level FFPM programming can get around some dramatically-wasteful cases that may arise in practice.

Having to copy large homogeneous data structures (arrays, for example) is the most problematic aspect of an FFPM's copying policy. Other approaches to this problem have tied it to the related problem of virtual memory and how to accommodate expressions too large for a Machine's L-array. Geoff Frank et al. [69; 70] reported several possible schemes. Common to the schemes is moving constant subexpressions out of the L-array into an addressable secondary memory and putting pointers to the backing store in the L-array. This relieves the L-array memory requirements, and sharing is possible by having many pointers to a single structure in secondary memory. The backing store can be managed in several ways. A conventional autonomous virtual-memory manager is one option. A more interesting attack is to embed virtual-memory actions into FFP programs; Frank et al. reported that this approach "makes much more sense for these [FFP] languages than for von Neumann languages" [70, page 8.44].

The second issue in sharing is computation sharing, to avoid the proliferation of unevaluated redexes. Traditionally, an FFPM uses applicative-order evaluation, in which unevaluated redexes are never copied. But what if the Machine were used so that such redexes were copied? Again, its different philosophy comes into play.

A sequential interpreter evaluates redexes one by one, so the duplication of unevaluated redexes means longer running times. Simple normal-order tree reduction of the λ-calculus, with its likely exponential blow-up of unevaluated redexes, is therefore unacceptable. An FFPM, on the hand, has a high processor-to-memory ratio (one processor per memory cell), and the presence of many redexes is not problematic, provided they can be reduced in parallel and they do not offend by simply taking up memory. (Because normal-order reduction is inherently sequential, an FFPM's abundance of processing power would not help; however, the point is valid for more practical λ-calculus systems.)

In cases where pointer-style sharing is critical, it may be simulated in an FFPM's low-level software. Magó did this in his algorithm for Paterson-Wegman unification on an FFPM [143].

5.1.12 Related non-graph-reduction architectures

Most parallel architectures to support functional programming use graph reduction; that work is reported in Section 3.5. Work on "closure-based" reduction and other techniques to speed up graph reduction is also reported there.

Besides the FFPM effort, the GMD Reduction Machine project was the other long-time advocate of string reduction to support functional languages. The design motivations that Berkling gave in his 1975 paper [26] are similar to the stated FFPM philosophy [144]. Both have their roots in Backus's early work on reduction languages [13].

The original GMD Reduction Machine [28; 124; 100; 126; 29] was a sequential computer; notably, it was the first reduction machine "to be successfully implemented" [125]. In my review here, I examine the parallel "cooperating reduction machines" described by Kluge [125]; the individual machines are the same as the sequential one. Each machine has three stacks, one of which initially holds a linearized, preorder representation of a λ-term. A search for a reducible expression proceeds by a railyard-like shunting of symbols between the stacks, until the right symbols appear at the top of the right stacks; then an unadulterated string-replacement reduction takes place. Expression shunting then resumes ... Parallelism develops by peeling off subterms and passing them for reduction to another "virtual machine." A virtual machine is mapped onto a real one if it can get a "ticket;" there are a limited number of tickets, providing a throttle on overexuberant parallelism.

The considerable data movement required by shunting terms around had its costs: Hommes says that "applications running on Berkling's reduction machine have shown that it is less suited for programs operating on large data structures" [99]. Hommes's solution was to move large data structures into a "heap," leaving behind appropriate pointers into the heap, and to have variants of data manipulation primitives (*head*, *tail*, etc.) that operate directly in the heap. (Note the similarities to the work of Frank et al. on the same problem for an FFPM [69; 70].) Schmittgen acknowledges a "severe performance problem" due to "the absence of, among other things, suitable abstract data types that support the efficient manipulation of non-atomic typed objects" [181]. In other words, having to build all data types out of primitive list structures proves expensive; therefore, Schmittgen proposes new built-in types for vectors, matrices and trees. She retains Hommes's ideas about special heap-twiddling operators.

Both the GMD machine and the FFPM à la Frank et al. exemplify an approach to copying and sharing: the underlying computational model unabashedly favors copying (pure tree reduction), but extra primitives to handle large aggregate data structures are provided for the occasions when copying would be most painful. The handling of aggregate data structures is a big issue in functional programming, generally [113; 98; 101; 10]. One can imagine that if this particular problem were solved, then sharing-in-general would indeed be unnecessary.

Other early string reduction machines include Treleaven and Hopkins's machine [198], which was inspired by Wilner's "recursive machines" [212]. The ZAPP (Zero-Assignment Parallel Processor) architecture [184; 148] is a "virtual tree machine" [43; 42]. ZAPP is similar to the parallel GMD in that tasks hand off subtasks to other processors; however, ZAPP focuses on divide-and-conquer algorithms in which the tasks spawn into a "process tree;" it depends on this "treeness" for success.

Perhaps the easiest kinds of parallel machine to build are Single-Instruction-stream, Multiple-Data-stream (SIMD) machines, in which processor-memory elements work in lock-step, obeying a single stream of instructions from a controller. One such architecture intended to support functional programming is John O'Donnell's Applicative Programming System Architecture (APSA); he has built a VLSI prototype and an emulator running on the better-known SIMD machine, the NASA MPP [159; 160; 158]. Though targeted at functional languages and made from two kinds of VLSI cells connected in a tree network, an APSA is fundamentally different from an FFPM. The language interpreter runs on a conventional host; the main purpose of the SIMD processor/memory is to avoid or speed up the manipulation of complex linked data structures endemic to functional language implementations (consider LISP lists and garbage collection, for example). Insofar as possible, linked structures are flattened into CDR-coded-like "compact linear structures" whose elements can be operated on concurrently by SIMD processors. In other words, an APSA also tries to improve overall performance by enhancing its data-structure operations.

The Connection Machine (CM) is a commercially-available SIMD computer with 65,536 processors and initial applications in image analysis and AI [93; 95; 94]. Unlike an APSA, it has a rich interconnection network (hypercube versus tree) and is application-driven (e.g., image analysis or AI), not language-driven. Nonetheless, a Common LISP variant, *Lisp, is a major programming language for the Machine; Steele and Hillis cite APL and FP as kindred languages [195]. They introduce a new data aggregate, the *xapping*, an unordered set of (*index*,*value*) ordered pairs: the index identifies a processor, the value is data at the processor; the CM then does operations on xappings in parallel. Whereas O'Donnell's data structure tricks are low-level and presumably invisible to the programmer, Steele and Hillis ask programmers to go halfway and re-code their programs the xapping way.

Again, notice the emphasis on improving aggregate data structures.

Hudak and Mohr looked at combinator-based graph reduction as a computational model to support functional languages on a CM [104]. They found a set of seven micro-operations ("graphinators") from which CM implementations of the Turner combinators [199] could be composed. The program graph is spread out in the CM processors, and a language interpreter is then a program that repeatedly pumps instructions for the graphinators at the processors. Though the idea is noteworthy and Hudak and Mohr go on to suggest refinements to the basic idea, they concede that they "must look more deeply for performance improvements." They conclude, "Although faster architectures can help, we feel that preserving locality of reference is the crucial line for future research" [104, page 233].

Another way to characterize an FFP Machine is as a *concurrent term rewriting* machine. Plaisted examined this aspect of his extended FFPM with a richer network; please see Section 5.4 for further discussion of his work [171]. An architecture that has taken the term-rewriting view from the beginning is the Rewrite Rule Machine (RRM) at SRI, under Goguen's direction [134; 79; 133; 135; 80]. Their basic programming language is the "ultra high-level" OBJ. At the lowest level, a "processor" is a controller plus an ensemble of SIMD cells that hold one-or-a-few tree nodes apiece. The processors are grouped into "clusters" that share an address space, and the clusters are joined into "networks". As best I can tell, the RRM does full-blown tree reduction; however it does use "multiplexed physical mapping" of nodes to somewhat-larger-grain cells and a scheme of "virtual pointers" in which substitutions need not be done immediately. In other words, it does copying but not necessarily as soon as the substitution is called for.

We have seen that the tack taken by almost all not-explicitly-graph reduction machines is to pay the copying price and to recoup the most profligate consequences with extra support for special data structures amenable to the machines' architectures and intended use.

5.2 An implementation of a λ_s-interpreter

This section gives the algorithms for an FFPM implementation of the λ_s-interpreter in Chapter 4. The only change is that reduction is to full β-normal form (BNF). Strangely enough, detecting if subterms are inside λ_s-abstractions is a mill-stone around an FFPM's neck, making reduction to weak β-normal form (WBNF) less desirable. Because this "implementation" serves mainly to provide space and time complexities for Sections 5.3.1 and 5.3.2 and because the comparison is on a per-step basis, the difference in the number of steps to reduce to BNF vs. WBNF is not important.

5.2.1 Basic algorithms

This section presents basic FFPM algorithms for ubiquitous tasks in a λ_s-interpreter implementation. All the algorithms are very low-level, usually laborious straight-line code. I have provided examples for the fascinated reader. Recall that First_i and Last_i bits are shown as (and) symbols on the Sel_i numbers, respectively. All asymptotic complexities are for the *worst case*. A is the number of cells in the "address space," i.e., the whole Machine; n is the number of cells in the λ_s-term or L-segment of interest. The footnote on page 108 explains why $\lg A$ factors in the worst case become $\lg n$ factors in the expected case on an FFPM.

Algorithm 5.1 Detecting bound variables of a given binder.

Given a λ_s-term of n cells in an L-segment, RIXs (relative indices) and RBLs (relative binding levels) calculated, we want to identify all the bound variables of a particular binder. Cells holding the left (cell L) and right (cell R) delimiters for the binder in question know their role. For example, given the example below, an attempt to find the bound variables in the subterm with cells L and R marked by $\Downarrow$ will identify the variables in cells marked by $\uparrow$. Some useful action would presumably follow this marking. Detecting bound variables is very common.

			$\Downarrow$																$\Downarrow$			
	$\{_x$	(	$\{_y$	(	(	x_2	y_1	)	(	$\{_z$	(	y_2	z_1	)	}	a_*	)	)	}	z_*	)	}
RIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
RBL:	0	1	1	2	2	2	2	2	2	2	3	3	3	3	2	2	2	2	1	1	1	0
							$\uparrow$					$\uparrow$										

1. Cell L broadcasts its RIX; cells with smaller RIX know they are not a bound variable. In the example above, this excludes cells 1–2. This step takes $O(\lg A)$ time.

2. Cell R broadcasts its RIX; cells with larger RIX know they are not a bound variable. This excludes cells 20–22. This takes $O(\lg A)$ time.

3. Cell L broadcasts its RBL, '1' in the example; call it top_bl. This takes $O(\lg A)$ time.

4. A cell holding a variable with binding index bi = RBL − top_bl is a bound variable. (Recall that variables' binding indices are shown as subscripts in the figures.) In the example, cells marked $\uparrow$ meet this condition. Other cells are not bound variables. This step takes $O(1)$ time.

Overall, it takes $O(\lg A)$ time (and no extra space) to detect a binder's bound variables.

Comments. Intuitively, a binding level counts binders, moving from the root to the leaves of a λ_s-tree. Subtracting top_bl adjusts the count to start from the binder in question instead of the root of the whole λ_s-term. A binding index, on the other hand, is counting binders from the its leaf position upwards. Where the two counts are equal, voilà!—a bound variable.

The ability to use a simple RBL scan as the basis for an $O(\lg A)$ algorithm to detect bound variables is my main reason for using binding indices.

Algorithm 5.2 Detecting free variables.

The actions to detect free variables in an n-cell λ_s-term are similar to those for detecting bound variables (Algorithm 5.1), with different tests performed in the individual cells. I presume we want to detect free variables with binders, not constants (variables free at the top level, with negative binding indices). As before, cells holding the left (cell L) and right (cell R) delimiters for the binder in question know their role. In the following example, there is only one free variable in the subterm set off by $\Downarrow$'s; it is marked with a $\uparrow$. (Steps 1–3 of this algorithm are identical to those of the preceding one.)

$$\begin{array}{rcccccccccccccccccccccc}
 & & & \Downarrow & & & & & & & & & & & & & & & & \Downarrow & & & \\
 & \{_x & (& \{_y & (& (& x_2 & y_1 &) & (& \{_z & (& y_2 & z_1 &) & \} & a_* &) &) & \} & z_* &) & \} \\
\mathsf{RIX:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 \\
\mathsf{RBL:} & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 0 \\
 & & & & & & \uparrow & & & & & & & & & & & & & & & &
\end{array}$$

1. Cell L broadcasts its RIX; cells with smaller RIX know they are not in the interesting subterm. In the example above, this excludes cells 1–2.

2. Cell R broadcasts its RIX; cells with larger RIX know they are excluded (cells 20–22).
3. Cell L broadcasts its RBL; call it top_bl. In the example, '1' is broadcast.
4. A cell holding a variable with binding index bi > RBL − top_bl is free. Other variables are bound or constants.

As for detecting bound variables, the algorithm takes $O(\lg A)$ time and uses no extra space.

Algorithm 5.3 Substitution.

I assume that the substitution $M[x := N]$ appears as a suspension to be completed, $[_x M\ N]$. We want to substitute N for the bound variables of the suspension in M (well-formedness precludes bound variables in N). I further assume that the suspension to be completed is alone in an L-segment of n cells, and that RIX, RBL, Sel_1, First_1, and Last_1 are known.

The algorithm proceeds in three phases: (1) mark bound variables of the suspension $[_x M\ N]$ and free variables in N, (2) copy N in place of each detected bound variable, and (3) adjust binding indices.

Mark variables. We need to find bound variables (in the suspension's body) and free variables in the suspension's pointee. Algorithms 5.1 and 5.2 may be used to look for bound variables and free variables everywhere in the suspension; to limit ourselves to free variables in the pointee, those detected by Algorithm 5.2 but with $\mathsf{Sel}_1 = 1$ (the body) unmark themselves. This phase takes $O(\lg A)$ time.

An example with ⇑'s marking bound variables and ⇓'s marking free variables in the pointee:

$$\begin{array}{rcccccccccccccccccccc}
 & & & & & & & & & & & & & & & & \Downarrow & & & \\
 & [_x & (& x_1 & \{_y & (& (& x_2 & y_1 &) & z_3 &) & \} &) & \{_x & (& x_1 & z_3 &) & \} &] \\
 & & & \Uparrow & & & & \Uparrow & & & & & & & & & & & & & \\
\mathsf{RIX}: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
\mathsf{RBL}: & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 & 0 \\
\mathsf{Sel}_1: & 0 & \langle 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2 & 2 & 2\rangle & 0
\end{array}$$

Copy.

1. Determine the size of the pointee ($\mathsf{Sel}_1 = 2$) and broadcast: the pointee cells that are First_1 or Last_1 send their RIX, and cells with bound variables (marked $\Uparrow$ above) take the difference between the two numbers (call it d). It takes $O(\lg A)$ time.

2. Each marked bound-variable cell asks to be cloned $d - 1$ times during storage management. In the worst case, $\frac{1}{2}n$ cells would each be cloning $\frac{1}{2}n$ copies of themselves, which is $O(n^2)$ clones total. Storage management follows; it is a linear process, so it takes $O(n^2)$ time. With clones marked by $\downarrow$, our example becomes:

```
            ↓  ↓  ↓  ↓  ↓                ↓  ↓  ↓  ↓  ↓
      [x (x1 x1 x1 x1 x1 x1 {y ( ( x2 x2 x2 x2 x2 x2 y1 ) z3  ) }  ) {x  (x1 z3  ) }  ]
RIX:  1  2  3  3  3  3  3  3  4 5 6  7  7  7  7  7  7 8 9 10 11 12 13 14 15 16 17 18 19 20
RBL:  0  1  1  1  1  1  1  1  1 2 2  2  2  2  2  2  2 2 2  2  2  1  1  1  2  2  2  2  1  0
Sel1: 0 ⟨1  1  1  1  1  1  1  1 1 1  1  1  1  1  1  1 1 1  1  1  1  1⟩ ⟨2  2  2  2  2  2⟩  0
```

3. Each marked-bound-variable clone cell saves its binding index in old_bi, to remember the binding distance up to the suspension. It takes $O(1)$ time.

4. Each cell in the pointee ($\mathsf{Sel}_1 = 2$) sends its symbol and whether it is marked as free (use RIX as a sort key, to keep them sorted). Each clone cell i overwrites itself with the i^{th} symbol received and saves the markedness in pteeMarked. This copying takes $O(n)$ time. Our example is now:

```
      [x ( {x (x1 z3 ) } {y ( ( {x (x1 z3 ) } y1 ) z3  ) }  ) {x  (x1 z3  ) }  ]
RIX:  1  2 3 3 3 3 3 3 4 5 6 7 7 7 7 7 7 8 9 10 11 12 13 14 15 16 17 18 19 20
RBL:  1  1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2  2  2  1  2  2  2  2  2  2  1
Sel1: 0 ⟨1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  1  1  1⟩ ⟨2  2  2  2  2  2⟩  0
```

Adjust binding indices. Variables that were free in the pointee (those with pteeMarked set) may now be at a greater binding depth and may need their binding index increased to match. The adjustment is: bi := bi + old_bi −2. All other free variables must then be decremented by one.

The suspension is now useless, so the cells holding its delimiters and the pointee erase their contents. As free variables were determined in the marking phase, all operations in this step take $O(1)$ time. The final result for the example is:

	(	$\{_x$	(	x_1	z_3	)	}	$\{_y$	(	(	$\{_x$	(	x_1	z_3	)	}	y_1	)	z_3	)	}	)
RIX:	2	3	3	3	3	3	3	4	5	6	7	7	7	7	7	7	8	9	10	11	12	13

In the worst case, a full-fledged substitution takes $O(n^2)$ time and space. For this reason, you will not see any full-fledged substitutions in this chapter!

Algorithm 5.4 Finding a matching syntactic delimiter.

It often arises that we have identified a cell holding an interesting delimiter (the left parenthesis of a redex, for example) and we need to identify the matching delimiter (e.g., its right parenthesis).

This is a simpler problem than having several delimiters marked and finding all of their matching delimiters in parallel. The simple case is adequate for the algorithms in this chapter.

I assume that a λ_s-term of n cells occupies an L-segment. A left delimiter is marked as being the symbol to be matched. (To match a right delimiter, do everything backwards.) Nesting levels (RNL) are precalculated. For example, in the λ-term shown, we want to match the symbol marked with $\downarrow$.

						$\downarrow$								
	(	(	z_*	z_*	)	$\{_x$	(	x_1	$\{_y$	y_1	}	)	}	)
RIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
RNL:	0	1	2	2	1	1	2	3	3	4	3	2	1	0

1. The marked left-delimiter cell broadcasts its RNL, the nesting level of interest. It takes $O(\lg A)$ time.

2. Do an integer-addition left-to-right scan: the marked left-delimiter contributes Restart-1, other left-delimiters on the playing nesting-level contribute Restart-0, and other cells contribute +0. The scan takes $O(\lg A)$ time. Our example, with cells' contributions to the scan, and its results:

						$\downarrow$								
	(	(	z_*	z_*	)	$\{_x$	(	x_1	$\{_y$	y_1	}	)	}	)
input:	+0	R0	+0	+0	+0	R1	+0	+0	+0	+0	+0	+0	+0	+0
result:	0	0	0	0	0	0	1	1	1	1	1	1	1	1
RIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
RNL:	0	1	2	2	1	1	2	3	3	4	3	2	1	0

3. Right delimiters on the interesting nesting level examine the result of the scan that they receive. The one receiving a 1 is the matching right delimiter. This takes $O(1)$ time.

Overall, it takes $O(\lg A) + O(\lg A) + O(1) = O(\lg A)$ time and no extra space to find a matching delimiter.

5.2.2 Controlling the interpreter

We now move from low-level algorithms to the top-level routines that control an FFPM implementation of a λ_s-interpreter. As usual, for each reduction step, we must *search* for a redex, *copy* the shared rator (if applicable), do the *β_s-reduction*, and *tidy* up. I explain the overall control of the reduction process first and then develop the algorithms in a top-down way.

A Front-End processor controls an FFPM's operation, driving a four-phase cycle of: global checking, partitioning into L-segments, executing reduction-routines in the active segments, and doing global storage management. (A traditional FFPM does not have a global-checking phase.) Partitioning is as described in Section 5.1.3, and storage management is unmodified from that in published FFPM descriptions [147]. Both the global checking and reduction-routine execution involve L-cells operating cooperatively on the λ_s-term. The global code always operates on the whole λ_s-term (using absolute level numbers), whereas reduction-routines execute in partitioned L-segments (using relative level numbers). Most work takes place in the L-segments, and I first review the control of what reduction routine is executed when.

Control flow in this FFPM implementation of the λ_s-interpreter is quite different from a conventional interpreter that controls its progress (basically) by pushing and popping function addresses on and off a stack. In an FFPM, the basic mechanisms are *associative matching* of symbol-patterns in cells and the *marking* (and unmarking) of delimiters. For example, the two-cell pattern $\boxed{(}\boxed{\{}$ indicates a β_s-redex; marking the left parenthesis and its matching right parenthesis would give "control" to that λ_s-application. As Section 5.1.3 describes, it is innermost marked symbol strings that are active and that execute reduction-routines.

The marking of an inner delimiter-pair in an L-segment, giving control to that substring of symbols, is analogous to a subroutine call. Conversely, when an L-segment unmarks its own delimiters so that the symbol-string enclosing it becomes active, it is analogous to a return from a subroutine call.

I also use an FFPM technique that I call *task hints*. In addition to marking an inner subterm's delimiters, a reduction-routine may insert a "task hint" anywhere in the inner subterm to suggest what is expected of it. The first thing a newly-active L-segment does is look for this hint (with associative matching). Task hints are *not* part of the λ_s-term; they are not strictly necessary but convenient.

Given a λ_s-term T to reduce to BNF, we first wrap it in an "envelope" with marked delimiters and the hint *reduce_to_BNF*:

$$(\textit{reduce\_to\_BNF}\ T)$$

This "term" will get control initially and whenever all the delimiters in T become unmarked.

5.2.3 Reducing to β-normal or root-lambda form

The procedure described here—looking for a redex and detecting when a term is in BNF—is ubiquitous in this implementation of the λ_s-interpreter. Its variant, reducing to root-lambda form (RLF), does not attempt reductions inside the top λ_s-abstraction and is used only for shared rators. Between them, these variants handle searching for redexes (with help from global checking, Section 5.2.6) and copying of shared rators. An L-segment chooses between the variants depending on the task hint found: *reduce_to_RLF* or *reduce_to_BNF*.

If the top-level "envelope" gets control, it tries to reduce to BNF. If an L-segment holding a suspension has control, it is understood that the *pointee* is to be reduced.

Algorithm 5.5 Reduce a λ_s-term to BNF by repeatedly finding its "leftmost" redex. This algorithm is executed if a *reduce_to_BNF* task hint is present.

The global-checking phase of the machine cycle (Section 5.2.6) will ensure that all cells to the left of the rightmost Follow*Fill[6] λ_s-pointer's target are not examined. If such a pointer exists, the search will take place in the pointee of the target suspension.

1. Look for the leftmost Ptr-marked λ_s-pointer (one of the two-cell associative matches $\boxed{(}\boxed{\hat{x}_i}$, $\boxed{\{}\boxed{\hat{x}_i}$, or $\boxed{\hat{x}_i}\boxed{)}$) or the leftmost redex (a two-cell

[6]The name Follow*Fill means "either FollowFill or FollowNoFill."

match, $\boxed{(}\boxed{\{}$). Algorithm 5.7 says how to do the matching and gives an example.

If the leftmost match is $\boxed{(}\boxed{\hat{x}_i}$, make it a FollowFill pointer $\acute{x}_i$ and drop a *reduce_to_RLF* hint. If the leftmost match is $\boxed{\{}\boxed{\hat{x}_i}$ or $\boxed{\hat{x}_i}\boxed{)}$, make it a FollowFill pointer $\grave{x}_i$ and drop a *reduce_to_BNF* hint.[7] In either case, include a *find_ptr_target* hint, so the next global-checking phase (Section 5.2.6) will ensure that the correct suspension gets control.

If the leftmost match is a redex $\boxed{(}\boxed{\{}$, then the matching right parenthesis needs to be found (Algorithm 5.4). Both parentheses will be marked, giving control to that λ_s-application. A β_s-reduction and tidying will follow, as Section 5.2.4 describes.

2. If Step 1 set up a Follow*Fill pointer, global checking (Section 5.2.6) will pass control somewhere else.

3. If Step 1 marked an inner λ_s-application, the current L-segment will be inactive in the next cycle.

4. If Step 1 found nothing, then this L-segment's job is done.

   If this segment is the outermost "envelope," then the whole reduction is complete.

   If this segment is a suspension, then its pointee is reduced and the bound variables in its body may be marked Followed. Algorithm 5.1 can be used to detect the bound variables and changing their hats is straightforward. Finally, erase the *reduce_to_BNF* hint and un-mark the suspension.

Comment. Recall the imagery of Figure 4.9 and the "life cycle" of a λ_s-pointer: plain-hatted pointers $\hat{x}_i$ slowly become Followed-hatted pointers $\bar{x}_i$ in a generally left-to-right sweep. The algorithm above is the main one responsible for the pattern.

As already mentioned, global-checking ensures that the matching takes place in the pointee of the target of the rightmost λ_s-pointer being followed (if one exists). Also, the parts of a λ_s-term that are already completely Followed simply will not generate a match. Finally, there is no need to worry that "indiscriminate" matching here will select something inside a suspension's pointee that has not been followed into. Since the overall term is tidy, there

[7]Actually, the variable mark could serve as the task hint.

would have to be a λ_s-pointer aimed at the pointee and it would be to the left. If that were the case, the leftmost match would be there, not in the pointee.

The following algorithm is applied when a suspension has a FollowFill λ_s-pointer aimed at it. It reduces the pointee only until it is a λ_s-abstraction, then substitutes it for the FollowFill pointer.

If the pointee does not reduce to a λ_s-abstraction but all the way to BNF, then all the suspension's bound variables (including the FollowFill) need to be re-hatted as Followed.

Algorithm 5.6 Reduce a λ_s-term to RLF. This algorithm is executed if a *reduce_to_RLF* task hint is present.

1. If the pointee is a λ_s-abstraction, it needs to be substituted for the FollowFill λ_s-pointer (there will only be one). A variant of full substitution (Algorithm 5.3) may be used, with obvious adjustments: only the FollowFill bound-variable is substituted for, and the suspension itself is not deleted, as other bound variables may remain.

   If the suspension λ_s-term occupies n cells, the pointee can be of size $O(n)$, so the new space for the one substitution is also $O(n)$. Storage-management time is proportional to the number of cells requested, so the substitution time is also $O(n)$. (That there is only one bound-variable being substituted keeps it from taking $O(n^2)$ time and space.)

2. If a FollowFill pointer was just filled and that pointer was the suspension's last bound variable, the suspension is now useless. Therefore, a check for useless suspensions (Algorithm 5.9) is in order.

3. If the pointee is not a λ_s-abstraction, do one step of reducing (the pointee) to BNF (Algorithm 5.5). That algorithm changes bound-variables' hats to Followed if it finds nothing.

4. If the FollowFill pointer was filled (Step 1) or the pointee eventually reduced to a non-λ_s-abstraction BNF, erase the *reduce_to_RLF* hint and unmark this suspension.

Comment. The $O(n)$ time and space complexities for filling a FollowFill pointer are the only non-$O(\lg A)$ complexities in this implementation of the λ_s-interpreter. They are directly attributable to the data-movement capabilities of the underlying tree networks. If data movement were improved by a richer network, these copying complexities would benefit.

Algorithm 5.7 Find the leftmost two-cell match.

The pattern $\boxed{(}\!\boxed{\{}$ indicates a redex. The pattern $\boxed{(}\!\boxed{\hat{x}_i}$ indicates a pointer-variable that is the rand of an application (thus, needs to be filled). The patterns $\boxed{\hat{x}_i}\!\boxed{)}$ and $\boxed{\{}\!\boxed{\hat{x}_i}$ indicate pointer-variables that need to be followed but not filled. All patterns appear in two adjacent cells.[8]

1. To detect any two-cell pattern in a term, the cells do a shift-right scan operation,[9] each sending its symbol; an example follows. The shift takes place in the pointee only.

$$\begin{array}{rcccccccccccccccccc}
 & [_y & (& \bar{x}_* & \grave{y}_1 &) & (& (& \bar{z}_* & \hat{w}_4 & &) & (& \{_x & x_1 & \} & \bar{z}_* &) &) \;] \\
\textsf{RIX:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\
\text{rotate:} & & & & & & & (& (& \bar{z}_* & \hat{w}_4 &) & (& \{_x & x_1 & \} & \bar{z}_* &) &) \\
 & & & & & & & & & & \uparrow & & \uparrow & & & & & &
\end{array}$$

2. Each cell examines what came from its left and decides if the two cells fit one of the patterns; the $\uparrow$ cells above are matches. Cells that match send their RIX into a Min single-result communication wave. In the example, the minimum RIX will be 10.

3. The right cell of the leftmost match will receive its RIX back. It reports where it is and what pattern was matched with a broadcast operation. Cell RIX $= 10$ will make itself into $\grave{w}_4$ and report a FollowNoFill match.

Each of the constituent operations takes no more than $O(\lg A)$ time.

[8]Intervening empty cells are taken care of by the FFPM hardware.
[9]There is an example on page 110.

5.2.4 β_s-reduction and local tidying

If a λ_s-application gains control, it is a redex. Algorithm 5.8, which implements the β_s rule, may be applied immediately. The suspension that takes the λ_s-application's place should stay marked, so it retains control.

A β_s-reduction may cause the need for tidying. Taking a "tidy while we are in the neighborhood" approach (rather than a hardwired "tidying phase" somewhere in the overall reduction cycle), I now consider the *local tidying* that can be done without the new suspension relinquishing control.

The local tidying rules that are tried following a β_s-reduction are the removal of a useless suspension (Algorithm 5.9) and the [triv-body] and [triv-ptee] rules, to remove a trivial suspension (Algorithms 5.10 and 5.11). These checks are mutually exclusive and may be applied sequentially.

Tidying that may involve higher-up parts of the λ_s-term is *nonlocal tidying*; global checking takes care of it. Sections 5.2.5 and 5.2.6 address nonlocal tidying and global checking, respectively.

The newly-created suspension (assuming it has not become useless and therefore nonexistent) may now be unmarked.

Algorithm 5.8 This algorithm implements the β_s rule.

The β_s rule is $(\lambda x.\{B\}\ N) \to [_x B\ N^{\uparrow\!f}]$. Unsurprisingly, its implementation is closely kin to that of substitution (Algorithm 5.3). I assume Sel_1, First_1, and Last_1 are pre-computed.

1. The algorithm begins by detecting the rator's bound variables and the free variables in the rand; this works the same as in substitution (taking $O(\lg A)$ time). Here is an example, with a bound variable marked with $\Uparrow$ and free variables (in the rand) marked with $\Downarrow$.

$$\begin{array}{lccccccccccccccccc}
 & & & & & & & & & & & & & & \Downarrow & \Downarrow & & \\
 & (& \{_x & (& \{_x & x_1 & \} & (& z_5 & x_1 &) &) & \} & (& z_4 & z_4 &) &) \\
 & & & & & & & & & \Uparrow & & & & & & & & \\
\mathsf{RIX}: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\mathsf{Sel}_1: & 0 & \langle 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2\rangle & 0
\end{array}$$

2. Bound variables in the rator get pointy hats; free variables in the rand are incremented by one. It takes $O(1)$ time, and gives:

$$\begin{array}{rccccccccccccccccc}
 & & & & & & & & & & & & & & \Downarrow & \Downarrow & & \\
 & (& \{_x & (& \{_x & x_1 & \} & (& z_5 & \hat{x}_1 &) &) & \} & (& z_5 & z_5 &) &) \\
 & & & & & & & & & \Uparrow & & & & & & & & \\
\mathsf{RIX:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
\mathsf{Sel}_1\mathsf{:} & 0 & \langle 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2\rangle & 0
\end{array}$$

3. Parentheses with $\mathsf{Sel}_1 = 0$ are changed to brackets; the λ-abstraction rator braces ($\mathsf{Sel}_1 = 1$ and either First_1 or Last_1 set) are erased. This takes $O(1)$ time, and gives the final result:

$$\begin{array}{rccccccccccccccc}
 & [& (& \{_x & x_1 & \} & (& z_5 & \hat{x}_1 &) &) & (& z_5 & z_5 &) &] \\
\mathsf{RIX:} & 1 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 13 & 14 & 15 & 16 & 17
\end{array}$$

Overall, the β_s rule takes $O(\lg A) + O(1) + O(1) = O(\lg A)$ time and no extra space.

Algorithm 5.9 Detect and remove a useless suspension (the [useless] pseudo-rule)

1. Algorithm 5.1 marks the bound variables in the suspension in $O(\lg A)$ time; there may not be any.

2. All the cells "vote" to determine if a bound variable exists (using a single-result operation).

3. If there are no bound variables, then (a) the free variables, detected with Algorithm 5.2, are decremented; and (b) the suspension brackets [ ] and all the cells in the pointee (Sel_1= 2) erase themselves.

All of the constituent operations take $O(\lg A)$ time in the worst case.

Algorithm 5.10 The [triv-body] rule.

The [triv-body] rule is applicable if the two-cell pattern $\boxed{[}\boxed{x_1}$ is present. It can be detected with a variant of Algorithm 5.7 in $O(\lg A)$ time.

Strictly speaking, the [triv-body] rule is a special case of substitution (Algorithm 5.3), but a much simpler algorithm may be used.

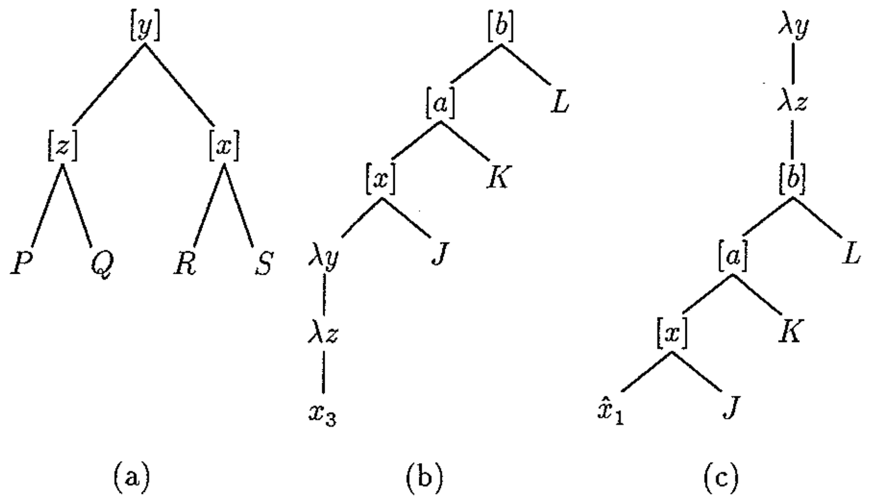


Figure 5.5: Non-local tidying

The trivial-body cell ($\mathsf{RIX} = 2$, a variable) erases itself and the free variables in the pointee (found with Algorithm 5.2) decrement themselves by one. The suspension delimiters (with $\mathsf{Sel}_1 = 0$) erase themselves.

The two communication operations (broadcasting, finding free variables in the pointee) take $O(\lg A)$ time; everything else takes constant time.

Algorithm 5.11 The [triv-ptee] rule.

The [triv-ptee] rule is applicable if the two-cell pattern $\boxed{x_i}\boxed{\|}$ is present. It can be detected with a variant of Algorithm 5.7 in $O(\lg A)$ time.

The [triv-ptee] rule is a special case of substitution and that algorithm (5.3, page 122) may be used. Because the pointee being copied occupies one cell (instead of $O(n)$ cells), no new space has to be allocated (no storage management costs) and the copying takes $O(\lg A)$ time.

5.2.5 Nonlocal tidying

If a β_s-reduction has just been done, it may have prompted the need for a [sus-rotl] tidying just above the new suspension. Figure 5.5a shows an example in which the newly-created $[x]$ suspension needs to be rotated upward.

Figure 5.5b shows an example in which the new $[x]$ suspension exposes the need to move the λy and λz abstractions above the *top* suspension $[b]$, as Figure 5.5c shows. These moves correspond to applying the [λ-up] rule *six times.* By gluing in more λ_s-abstractions and suspensions, an arbitrarily large number of [λ-up]'s might be needed. Fortunately, it is possible to have an FFPM reduction-routine that recognizes the symbol string

$$[^m\{^nB\}^n\ P_1]P_2]\cdots P_m]$$

(where $[^m$ corresponds to $\overbrace{[[[\cdots[}^{m\text{ of them}}$) and converts it into

$$\{^n[^mB'\ P_1']P_2']\cdots P_m']\}^n$$

(where the primes indicate terms with adjusted binding indices). This "[multi-λ-up] rule" (it is not really a rewrite rule) implements the potentially-long sequence of [λ-up]'s in one step.

5.2.6 Global checking

Global checking is a phase of the FFPM cycle that operates on the entire λ_s-term in the Machine regardless of delimiter markings; it uses absolute level numbers. The first problem it solves is nonlocal tidying, as just described. To this end, global checking runs Algorithms 5.13 and 5.14.

The second problem global checking solves comes from the marking-unmarking style of control that reduction-routines use. If the λ_s-pointer is deeply nested within marked delimiters and its target suspension is far above it in the λ_s-tree and that suspension must "get control," then it could take many cycles just to unmark all the delimiters so the target suspension would become innermost-marked and thereby gain control. If requested (by a *find_ptr_target* hint), global checking gives control to the suspension target of the rightmost Follow*Fill pointer in one step, using Algorithm 5.12.

Algorithm 5.12 Activate the target suspension of the rightmost Follow*Fill λ_s-pointer. This algorithm is executed only if a *find_ptr_target* task hint is present.

If one or more Follow*Fill λ_s-pointers exist (anywhere in the λ_s-term), then be sure that the suspension to which the *rightmost* one points is marked and that it has no marked delimiters within it (i.e., erase them).

1. To detect the rightmost Follow*Fill λ_s-pointer, a single-result Max operation on AIXs will locate the "interesting" symbol (non-pointer cells do not play). It takes $O(\lg A)$ time.

				↓				↓								↓									
	$[_y$	$[_x$	(	$\acute{x}_1$	$\hat{y}_2$	)	(	$\acute{y}_2$	z_*	)	]	(	$[_w$	(	z_*	$\dot{w}_1$	)	(	z_*	z_*	)	]	z_*	)	]
AIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
ABL:	0	1	2	2	2	2	2	2	2	2	1	1	1	2	2	2	2	2	2	2	2	1	1	1	0

In this example, the cells marked ↓ participate in the Max operation; cell 16 is selected.

2. The interesting cell, knowing its binding index and its (absolute) binding level (ABL) can calculate the binding level of its target suspension (ABL − binding index) and broadcast it, so the interesting suspensions know who they are. In the example, they have ABL = 1.

3. Next, do a left-to-right integer-addition scan: the interesting Follow*Fill pointer sends +1, the interesting-suspension right-brackets send a Restart-0, others do not play. The only interesting-suspension right-bracket that will receive a +1 is the target suspension. Algorithm 5.4 can then find the matching left-bracket. The example, with scan input and result shown (zeros excluded, the "hit" shown by ⇑):

	$[_y$	$[_x$	(	$\acute{x}_1$	$\hat{y}_2$	)	(	$\acute{y}_2$	z_*	)	]	(	$[_w$	(	z_*	$\dot{w}_1$	)	(	z_*	z_*	)	]	z_*	)	]
AIX:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
ABL:	0	1	2	2	2	2	2	2	2	2	1	1	1	2	2	2	2	2	2	2	2	1	1	1	0
input:											R0					+1						R0			
result:																	1	1	1	1	1	1			
																						⇑			

4. The brackets of the now-detected desired suspension must be marked and any cells that are between those brackets (just compare AIXs) need to be unmarked so the desired suspension will be innermost.

5. The *find_ptr_target* task hint will be erased.

Comment. All together, it takes $O(\lg A)$ time. The extension of using binding indices for λ_s-applications and suspensions, discussed in Section 6.7, could be used in conjunction with this global checking as a way to quickly restore the markings that this step may erase.

Algorithm 5.13 The [sus-rotl] rule.

The rule is: $[_x B\ [_y P\ Q]] \rightarrow [_y[_x B^{if}\ P^{ib}]\ Q^{df}]$. It is applicable if the two-cell pattern $\boxed{[[}$ is present; it can be detected with a variant of Algorithm 5.7 in $O(\lg A)$ time. The outer suspension is activated, then the algorithm proceeds as with any reduction routine. An example λ_s-term:

$$\begin{array}{lcccccccccccccccc}
 & [_x & (& \hat{x}_1 & z_2 &) & [_y & (& z_3 & \hat{y}_1 &) & (& z_3 & z_3 &) &] &] \\
\mathsf{RIX}: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\mathsf{Sel}_1: & 0 & \langle 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2\rangle & 0 \\
\mathsf{Sel}_2: & & & & & & 0 & \langle 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2\rangle & 0 &
\end{array}$$

1. Detect and increment the *free* variables in B (cells with $\mathsf{Sel}_1 = 1$); shown by $\downarrow$ in the example below. Algorithm 5.2 does this, in $O(\lg A)$ time.

   Similarly, detect and decrement the *free* variables in Q (cells with $\mathsf{Sel}_1 = 2$ and $\mathsf{Sel}_2 = 2$); shown by $\Downarrow$ in the example below.

   Also, detect and increment the *bound* variables of $[y]$ in P ($\mathsf{Sel}_1 = 2$ and $\mathsf{Sel}_2 = 1$); shown by $\Uparrow$ in the example. A slight variant of Algorithm 5.1 can do this in $O(\lg A)$ time.

$$\begin{array}{lcccccccccccccccc}
 & & & & \downarrow & & & & & & & & \Downarrow & \Downarrow & & & \\
 & [_x & (& \hat{x}_1 & z_3 &) & [_y & (& z_3 & \hat{y}_2 &) & (& z_2 & z_2 &) &] &] \\
 & & & & & & & & & \uparrow & & & & & & & \\
\mathsf{RIX}: & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\
\mathsf{Sel}_1: & 0 & \langle 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2\rangle & 0 \\
\mathsf{Sel}_2: & & & & & & 0 & \langle 1 & 1 & 1 & 1\rangle & \langle 2 & 2 & 2 & 2\rangle & 0 &
\end{array}$$

2. The cell holding $[_y$ erases itself ($\mathsf{Sel}_1 = 2$ and First_1 set). The last cell in P ($\mathsf{Sel}_1 = 2$, $\mathsf{Sel}_2 = 1$, and Last_2 set) clones one copy of itself and the right cell becomes a suspension right-delimiter. This takes $O(1)$ time.

   The first cell ($\mathsf{RIX} = 1$) clones one copy of itself. The last cell ($\mathsf{Sel}_1 = 0$ and holding a $]$) erases itself. Both operations take $O(1)$ time. The final result is ($\Downarrow$'s show rewritten cloned cells):

$$\begin{array}{lcccccccccccccccc}
 & & \Downarrow & & & & & & & & & \Downarrow & & & & & \\
 & [_x & [_x & (& \hat{x}_1 & z_3 &) & (& z_3 & \hat{y}_2 &) &] & (& z_2 & z_2 &) &] \\
\mathsf{RIX}: & 1 & 1 & 2 & 3 & 4 & 5 & 7 & 8 & 9 & 10 & 10 & 11 & 12 & 13 & 14 & 15 \\
\mathsf{Sel}_1: & 0 & 0 & \langle 1 & 1 & 1 & 1\rangle & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2\rangle \\
\mathsf{Sel}_2: & & & & & & & \langle 1 & 1 & 1 & 1\rangle & 1\rangle & \langle 2 & 2 & 2 & 2\rangle & 0
\end{array}$$

The [sus-rotl] rule takes $O(\lg A) + O(\lg A) + O(\lg A) + O(1) + O(1) = O(\lg A)$ time and no extra space (it gives up as much as it asks for in clones).

Comment. The creation of a new suspension by β_s-reduction can induce at most one firing of the [sus-rotl] rule. The removal of a suspension from the top of a pointee cannot expose further suspensions below it; for that to arise, the lower suspension would have to have been created earlier, but that is impossible with normal-order evaluation.

Algorithm 5.14 The [multi-λ-up] pseudo-rule: repeated applications of the [λ-up] rule.

The pseudo-rule is: $[^m\{^n B\}^n\ P_1]P_2]\cdots P_m] \rightarrow \{^n[^m B'\ P'_1]P'_2]\cdots P'_m]\}^n$. It is applicable if the two-cell pattern $\boxed{[\{}$ is present; it can be detected with a variant of Algorithm 5.7 in $O(\lg A)$ time. I am using an that is not tidy (it has trivial pointees), so it will fit on the page.

1. The tell-tale pair $\boxed{[\{}$ is in the larger pattern $\overbrace{[[[\cdots[}^{m\text{ of them}}\quad\overbrace{\{\{\{\cdots\{}^{n\text{ of them}}$.

   First, broadcast where the tell-tale left-bracket is (AIX $= 17$ below, marked $\downarrow$; presumably, there are 14 symbols to the left in the overall λ_s-term). Then, to find the leftmost bracket [and to calculate m, the *non*-left-bracket cells to the left of the initial [{ contribute their AIX to a Max operation; the result-plus-one is the index of the leftmost bracket. The leftmost non-left-bracket has AIX $= 14$, so cell 15 is marked ($\Downarrow$). The difference between that AIX and the initially-matched left-bracket's AIX is m. Here, $m = 17 - 14 = 3$.

$$\begin{array}{lccccccccccccccccc}
 & \Downarrow & & \downarrow & & & & & & & & & & & & & & \\
 & [_c & [_b & [_a & \{_x & \{_y & (& x_2 & b_4 &) & \} & \} & z_4 &] & z_3 &] & z_2 &] \\
\text{AIX:} & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\
 & & & & \uparrow & \Uparrow & & & & & & & & & & & &
\end{array}$$

   An analogous operation to the right finds the rightmost left brace { and determines n; the endmost braces are marked $\uparrow$ and $\Uparrow$ above. $n = 20 - 18 = 2$. These operations take $O(\lg A)$ time.

2. The matching right-bracket] of the leftmost left-bracket $\Downarrow$ can be found with Algorithm 5.4.

   The matching $\}^n$ right braces can be found by delimiter matching (Algorithm 5.4) and the repeated-symbol detection just used. The example, with all the markers now shown for the matching delimiters also, is:

$$\begin{array}{rccccccccccccccccc}
 & \Downarrow & & \downarrow & & & & & & & & & & \downarrow & & & & \Downarrow \\
 & [_c & [_b & [_a & \{_x & \{_y & (& x_2 & b_4 &) & \} & \} & z_4 &] & z_3 &] & z_2 &] \\
\mathsf{AIX:} & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\
 & & & & \uparrow & \Uparrow & & & & & \Uparrow & \uparrow & & & & & &
\end{array}$$

3. The leftmost $m+n$ cells—m left-brackets and n left-braces—reorder themselves to n left-braces followed by m left-brackets. Since the indices of interesting symbols and the numbers m and n are known from just-completed broadcast operations, this takes $O(1)$ time. This gives (changed cells marked $\downarrow$):

$$\begin{array}{rccccccccccccccccc}
 & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & & & & & & & & & & & \\
 & \{ & \{ & [& [& [& (& x_2 & b_4 &) & \} & \} & z_4 &] & z_3 &] & z_2 &] \\
\mathsf{AIX:} & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31
\end{array}$$

4. The $\}^n$ string of right-braces erases itself.

   The rightmost right-bracket clones itself n times, and the string of clones becomes the string $]\,\}^n$. This gives (interesting cells marked $\Downarrow$):

$$\begin{array}{rccccccccccccccccc}
 & & & & & & & & & & & & & & & \Downarrow & \Downarrow & \Downarrow \\
 & \{ & \{ & [& [& [& (& x_2 & b_4 &) & z_4 &] & z_3 &] & z_2 &] & \} & \} \\
\mathsf{AIX:} & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 26 & 27 & 28 & 29 & 30 & 31 & 31 & 31
\end{array}$$

5. It remains only to adjust binding indices.

   Variables in B that were bound to any of the $\{^n \cdots \}^n$ λ_s-abstractions need to have m added to them (shown by $\uparrow$ below). Variables in B that were bound to any of the m suspensions need to have n subtracted from them (shown by $\downarrow$ below).

   Free variables in the terms $P_1, P_2, \ldots, P_m$ need to be incremented by n, shown here by $\Uparrow$ in the final result.

$$\begin{array}{llllllllllllllllll}
 & & & & & & \uparrow & \downarrow & & & & & & & & & \\
 & \{ & \{ & [& [& [& (\,x_5 & b_2 &) & z_6 &] & z_5 &] & z_4 &] & \} & \} \\
\mathsf{AIX:} & 15 & 16 & 17 & 18 & 19 & 20\;21 & 22 & 23 & 26 & 27 & 28 & 29 & 30 & 31 & 31 & 31 \\
 & & & & & & & & & \Uparrow & & \Uparrow & & \Uparrow & & &
\end{array}$$

Variants of the algorithms to find bound variables (Algorithm 5.1) and free variables (Algorithm 5.2) that check for *ranges* of binding indices would be needed in this step.

The basic constituent operations takes more than $O(\lg A)$ time; the cloning in Step 4 can take $O(n)$ time, but that is a vestige of the simplified representation used here. This is considered further on page 149. The algorithm here uses no extra space (it gives up as much as it asks for in clones).

5.2.7 Summary of the FFP Machine implementation

Reduction of the recurring example. Figure 5.6 shows steps of the recurring example. You may wish to compare it with Figures 2.8 (plain tree reduction), 3.7 (graph reduction), and 4.20 (λ_s-trees shown).

Noteworthy aspects. The FFPM implementation of the λ_s-interpreter is notable in several ways.

1. It is controlled almost exclusively by searching for two-cells patterns using distributed associative-matching.

2. Every algorithm is straight-line code with no repetitions (loops).

3. Except for those in which λ_s-terms are copied, every algorithm takes $O(\lg A)$ time in the worst case, where A is the size of the address space.[10] Algorithms with copying take $O(n)$ time. No algorithm has a worse time complexity.

4. Implementing the λ_s-interpreter on a similar architecture but with a richer interconnection network would presumably improve the time complexity for copying and, therefore, the most "expensive" operations in this implementation.

[10]The footnote on page 108 says why we can *expect* $O(\lg n)$ execution, where n is the size of the term.

The initial term, with first redex marked:

$$\{_y \ \overset{\vdash}{(} \ \{_f \ (\ f_1 \ (\ f_1 \ (\ f_1 \ f_1 \ ) \ ) \ ) \ \} \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ \overset{\dashv}{)} \ \}$$

After one reduction and the next two-cell match (marked $\Downarrow$):

$$\{_y \ [_f \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\hat{f}_1} \ (\ \hat{f}_1 \ (\ \hat{f}_1 \ \hat{f}_1 \ ) \ ) \ ) \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After a lazy copy of the shared rator (copy marked $\downarrow$):

$$\{_y \ [_f \ (\ \overset{\downarrow}{\{_x} \ \overset{\downarrow}{(} \ \overset{\downarrow}{x_1} \ \overset{\downarrow}{y_3} \ \overset{\downarrow}{)} \ \overset{\downarrow}{\}} \ (\ \hat{f}_1 \ (\ \hat{f}_1 \ \hat{f}_1 \ ) \ ) \ ) \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After the 2nd reduction and two matches are made (both marked $\Downarrow$):

$$\{_y \ [_f \ [_x \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\acute{x}_1} \ y_3 \ ) \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\hat{f}_2} \ (\ \hat{f}_2 \ \hat{f}_2 \ ) \ ) \ ] \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After the 3rd reduction, before tidying (new suspension marked):

$$\{_y \ [_f \ [_x \ (\ \acute{x}_1 \ y_3 \ ) \ [_x \ (\ x_1 \ y_4 \ ) \ (\ \hat{f}_3 \ \hat{f}_3 \ ) \ ] \ ] \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After [sus-rotl], with two matches made (marked $\Downarrow$):

$$\{_y \ [_f \ [_x \ [_x \ (\ \acute{x}_1 \ y_4 \ ) \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\acute{x}_2} \ y_4 \ ) \ ] \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\hat{f}_3} \ \hat{f}_3 \ ) \ ] \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After lazy copy, 4th reduction, and tidy; then match made (marked $\Downarrow$):

$$\{_y \ [_f \ [_x \ [_x \ (\ \acute{x}_1 \ y_4 \ ) \ (\ \acute{x}_2 \ y_4 \ ) \ ] \ \overset{\Downarrow}{(} \ \overset{\Downarrow}{\hat{f}_2} \ y_4 \ ) \ ] \ \{_x \ (\ x_1 \ y_1 \ ) \ \} \ ] \ \}$$

After last-instance relocation, 5th reduction, and tidy:

$$\{_y \ [_x \ [_x \ (\ \acute{x}_1 \ y_3 \ ) \ (\ \acute{x}_2 \ y_3 \ ) \ ] \ (\ y_2 \ y_2 \ ) \ ] \ \}$$

After detecting that all FollowFill-pointer targets are reduced:

$$\{_y \ [_x \ [_x \ (\ \bar{x}_1 \ y_3 \ ) \ (\ \bar{x}_2 \ y_3 \ ) \ ] \ (\ y_2 \ y_2 \ ) \ ] \ \}$$

Figure 5.6: The FFP Machine interpreter on the recurring example

I doubt that a richer network would significantly improve the $O(\lg A)$ time complexity of the tree broadcast and scan operations.

FFPM algorithms are unusual in that storage management costs are not swept under the rug.

5. Most importantly, the implementation is not specifically wedded to an FFPM. Any architecture that would support the linear representation of λ_s-terms and provide fast scan operations could support this style of λ_s-interpreter.

6. The collected algorithms of this interpreter represent a fair sampling of low-level FFPM programming techniques, which are *fun*.

Chapter 6 considers further extensions to the λ_s-interpreter that could be important for a practical λ_s-style implementation.

5.3 Equivalence to graph reduction: efficiency

This section collects the worst-case asymptotic space and time complexities for the just-sketched FFPM implementation of a λ_s-interpreter and compares them with those of a graph-reducing λ_g-interpreter (Chapter 3) on a global-addressable-memory (GAM) machine. The goal is to show that a particular type of tree reduction of the λ-calculus—with suspensions—can compare favorably with graph reduction; matching its sharing properties is the key.

One cannot discuss time and space complexity of λ-calculus implementations without revealing the computational models used. Comparing implementations based on different models is not ideal. The problem is, of course, that neither of these interpreters would do well if implemented on the other computational model. Fortunately, the task here is not to show that one interpreter is x percent faster than the other, but to show that tree reduction, appropriately defined and with the right underlying primitives, is asymptotically comparable to graph reduction.

Because the normal-order evaluation of the λ-calculus is entirely sequential, the basic algorithm (search for a redex, copy the shared rator, ...) does not differ on a sequential or a parallel machine architecture. Though an FFPM wins hands down on some parts of a reduction step, I do not use this to offset another weakness because of its tree-reduction nature.

I reiterate that my complexity figures for the λ_s-interpreter are not irredeemably bound to an FFPM architecture; any machine that would support the linear representation of terms and provide fast scan primitives could likely do as well.

5.3.1 Space complexity

Theorem 5.1 *An implementation of the λ_g-interpreter (graph reduction) uses the same amount of space as an implementation of the λ_s-interpreter, within a constant factor. This presumes tidied λ_s-terms.*

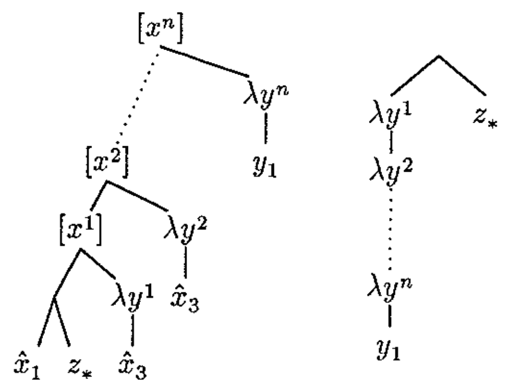
Proof. Given that Theorem 4.9 showed that a λ_g-interpreter and λ_s-interpreter maintain an exact correspondence between plain nodes through each reduction step, it suffices to show that the interpreters represent plain nodes in comparable space and that tidied λ_s-terms cannot become bloated by adding non-plain nodes.

Representing plain nodes. A plain node (λ-application, λ-abstraction, or a variable) can be represented in a constant amount of space. In a λ_g-interpreter, this would presumably include space for the pointers that represent edges in the λ_g-graph. In a λ_s-interpreter, the edges in a λ_s-tree are not explicitly represented, thanks to the linear representation. However, the nonterminal nodes represented by paired delimiters occupy *two* cells in the simple representation I have used. Middleton gives a thorough treatment of program representation techniques in an FFPM, and they would be applicable to the λ-calculus [152]. Help as they might, they would not change the $O(1)$ space requirement per plain node.

λ_s-term bloating. Besides plain nodes, a λ_s-term may also include suspensions and λ_s-pointers. Can these be added in such a way as to unbalance the equivalence between λ_g-interpreter and λ_s-interpreter for space to represent a λ-term? No, but untidied terms must be excluded— a λ_s-term can have arbitrarily many plain nodes if useless suspensions are hung all over it.

The worst possible ratio of non-plain to plain nodes in a tidied λ_s-term is in a term of the form shown in Figure 5.7a; Figure 5.7b shows the equivalent λ-term with plain nodes only. (The superscripts are there only to show how many nodes there are.) The "bloated" λ_s-term has $3n + 3$ nodes, and the plain equivalent has $n+3$ nodes—the difference is still only a constant factor.

In summary, a tidied λ_s-term that is equivalent to a λ_g-graph with n plain nodes will have at most $O(n)$ nodes (plain or otherwise). □



(a) (b)

Figure 5.7: A λ_s-term stuffed with non-plain nodes

Comment. Searching for a redex does not in any way alter the size of the λ-term in either a λ_g-interpreter or a λ_s-interpreter. That copying a shared rator, actually β-reducing the redex, or tidying up (in the λ_s-interpreter) always maintain an exact correspondence between plain nodes was discussed in Sections 4.2–4.5.

A minor note about last-instance relocation, which graph reduction does not do: this operation will use some space as it copies the pointee to its new place; however, it will be followed by the removal of a useless suspension, thus reclaiming an equal amount of space to that just used in the copy. If done sequentially, the λ_s-interpreter would suffer a temporary "bulge" that the λ_g-interpreter would not see; there is, however, no reason why the two operations (copying and reclaiming) could not be interleaved on an FFPM by an optimized reduction routine.

5.3.2 Time complexity

This section compares the time complexity of a λ_g-interpreter implemented on a GAM machine with that of a λ_s-interpreter implemented on an FFPM. Again, this analysis is based on Theorem 4.9, which says that the interpreters maintain an exact correspondence between plain nodes, step for step.

Time for one reduction step

Table 5.2 shows the collected worst-case asymptotic time complexities for

	λ_g-interpreter	λ_s-interpreter	
		FFPM:	FFPM:
phase of	GAM	simple	fancy
reduction step	machine	representation	representation
finding the next redex	$O(n \lg A)$	$O(n \lg A)$	same
copying a shared rator	$O(n \lg A)$	$O(n)$	same
copying a last instance	—	$O(n)$	same
β-reduction	$O(n \lg A)$	$O(\lg A)$	same
tidying	—	$O(n)$	$O(\lg A)$

Table 5.2: Comparison of asymptotic time complexities

the implementations of the two interpreters. In all cases, n is the number of nodes in the λ_g-graph or λ_s-term; A is the size of the "address space" for that model. This section explains the table and justifies its claim.

Memory access time. There exists an unfortunate custom of saying memory access in a GAM machine takes "constant time." Each access to such memory requires the use of an address-decoding tree and so, strictly speaking, takes $O(\lg A), A \geq n)$. This explains the "$\lg A$" factors in the GAM machine column in Table 5.2. In an FFPM, tree operations may travel the full height of the physical tree, depending on alignment, hence the $\lg A$ factors there.

While it is true that the constant factor is small in "log-time" memory access, the same might be true of the tree-network hardware in an FFPM that produces the "$\lg A$" factors in the FFPM column of Table 5.2. Either both must be charged the logarithmic factor (my choice) or neither should be.

Finding the next redex. In the worst case, a λ_g-interpreter must examine an entire n-node λ_g-graph before finding a redex (or not finding one). An example not quite so bad, but still taking $O(n \lg A)$ time, is shown in Figure 5.8a; normal-order evaluation must visit roughly $\frac{1}{2}n$ nodes in its pre-order walk of the graph before finding the redex $(\lambda x.\{x_1\}\ z_*)$. Each pointer-hop takes $O(\lg A)$ time.

Figure 5.8b shows one possible equivalent λ_s-term to the term in Figure 5.8a. This λ_s-term would also require the same sequential search through the (plain) λ_s-application nodes; the only difference is that the applications are connected by λ_s-pointers. Each of the $\hat{x}$ λ_s-pointers would be followed in turn: it would first be matched and marked as FollowFill (Algorithm 5.5)

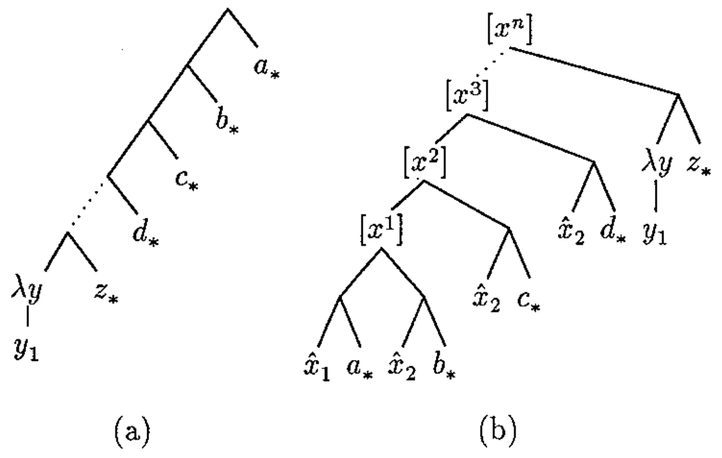


Figure 5.8: Unpleasant example of looking for redex

then, following global checking (Section 5.2.6), the search would continue at the suspension's pointee (Algorithm 5.6). All those algorithms take $O(\lg A)$ time. Therefore, it takes one FFPM cycle to follow one λ_s-pointer, in $O(\lg n)$ time. To traverse n λ_s-pointers (the worst case) will then take $O(n \lg A)$ time.

This $O(\lg A)$ time to follow a λ_s-pointer depends on λ_s-terms being tidied; it is *the* reason for tidying. Without the [triv-ptee] and [sus-rotl] rules applied, getting from one plain node to the next could involve following *many* λ_s-pointers, $O(n)$ of them in the worst case. Preventing this case is the reason for so constraining the kinds of λ_s-pointers allowed in a tidied term. Also, recall that the main matching algorithm for searching (Algorithm 5.7) depends on useless suspensions being removed.

Of course, a λ_s-interpreter on an FFPM will often find a redex much quicker than a sequential λ_g-interpreter. For example, Figure 5.8a is a perfectly good λ_s-term, and the searching algorithm (Algorithm 5.7) will find the redex $(\lambda x.\{x_1\}\ z_*)$ with one $O(\lg A)$ match.

Copying a shared rator. Before a β-reduction actually happens, if the rator is shared, then a new copy must be made. These interpreters use lazy copying; Section 4.4 describes the techniques, and Lemma 4.2 ensures that the λ_g- and λ_s-interpreters will copy corresponding plain nodes. Theorem 5.1 has already assured that non-plain-node bloating can increase the size of a λ_s-term only by a constant factor. If the whole term has n plain nodes, then in the worst case almost all of them will have to be copied, meaning $O(n)$ node copies. For both a GAM machine and an FFPM the time to copy is proportional to the size of the copy; strictly speaking, $O(n \lg A)$ on a GAM machine and $O(n + \lg A)$ on an FFPM.

The time to copy dominates the time to re-do the backpointers in the new term (in a λ_g-interpreter on a GAM machine) or to adjust free variables' binding indices (using Algorithm 5.2 in a λ_s-interpreter on an FFPM).

Last-instance relocation. As discussed in Section 4.4, if the λ-abstraction rator is *not* shared but *is* the target of a λ_s-pointer, then a λ_s-interpreter must move that "last instance" of the λ_s-abstraction into place before β_s-reduction can proceed. Since the copy is just like that of a shared rator, it can take $O(n)$ time.

Because graph reduction requires no comparable effort, this last-instance relocation is a blow to the step-for-step equivalence of a λ_s-interpreter to a λ_g-interpreter; it is the *only* such blow.

It stands to reason that a reduction scheme that uses a linear representation of terms and that requires any reduction rule to operate locally must sometimes pay the price and move symbols into a "local" position. In contrast, every location in a global addressable memory is equally accessible (if not sharable); symbols need not be moved to make access more convenient.

Several comments are in order. The first is that it is *only* λ_s-abstractions that are rators of redexes about to be reduced that will be relocated. λ_s-abstractions, representing functions or "code," are typically relatively small; large aggregate data-structures that are painful to move are unlikely to be relocated. Second, because the relocation works just like a shared-rator copy, the relocation stops when λ_s-pointers are reached—the parts of the λ_s-abstraction shared *before* the relocation are shared afterwards, too: they are not moved. This would further limit the size of relocations.

The amount of wasted copying due to last-instance relocation is bounded in the case of the λ-calculus-equivalent of "straight-line code" in which no sharing occurs. Figure 5.9a shows a λ_s-term with a repeated structure from which a term of arbitrary size could be constructed. Figure 5.9b shows that term after one reduction (to create $[x]$), just before the last-instance moving of the top pointee—practically the whole term will be moved. Figure 5.9c shows the result after the β_s-reduction and the removal of two useless suspensions $[x]$ and $[y]$, just before the next last-instance relocation. With this pattern, each last-instance relocation will move seven fewer (plain) nodes than the time before. If the λ_s-term starts with n nodes, the maximum extra copying due purely to last-instance relocation will be $n + (n-7) + (n-14) + \cdots$, that is, $O(n^2)$ extra node-copies. It takes a highly contrived example to achieve

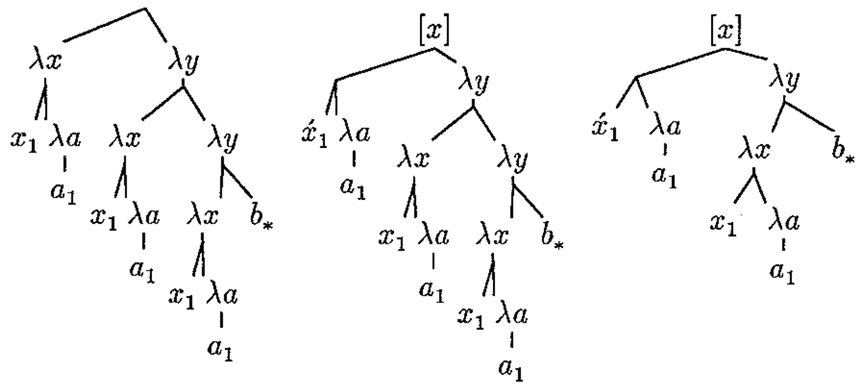


Figure 5.9: Last-instance relocation in straight-line code

this.

Though the amount of trouble last-instance relocation can cause *by itself* is bounded, such relocation is not bounded in general. Specifically, a subterm with a last instance to be moved can be embedded in a shared term that will be copied over and over (in graph reduction as well). Inside of each copy, there will be a last-instance relocation for the λ_s-interpreter to do. For example, in Figure 5.10 (a non-terminating reduction), each time the $\lambda s.\{\cdots\}$ subterm is copied, it carries what will become a last-instance relocation of $\lambda z.\{(z\ z)\}$. (The figure shows the first reduction, then omits two reductions, then shows the λ_s-term before and after the last-instance relocation and its reduction.) Again, it takes effort to find such examples.

Does the "tree reduction overhead" of last-instance relocation cripple this style of reduction versus graph reduction? Theoretically speaking, it certainly precludes their absolute equivalence in time complexity.

I suggest that last-instance relocation is a fair price to pay for a linear representation with strong locality. My experience and the far-fetchedness of the "ill-behaved" examples suggest that the λ_s-abstractions to be moved will be quite small. Moreover, λ-lifting techniques to remove (and share) free expressions at compile-time would tend to make them smaller still. Going further, Section 6.6 describes techniques that might ameliorate this cost, and Section 6.3 presents smarter suspensions one might use with a practical λ-calculus, in which the suspension would do (part of) the work for the redex proper.

The practical cost of last-instance relocation will only come from ex-

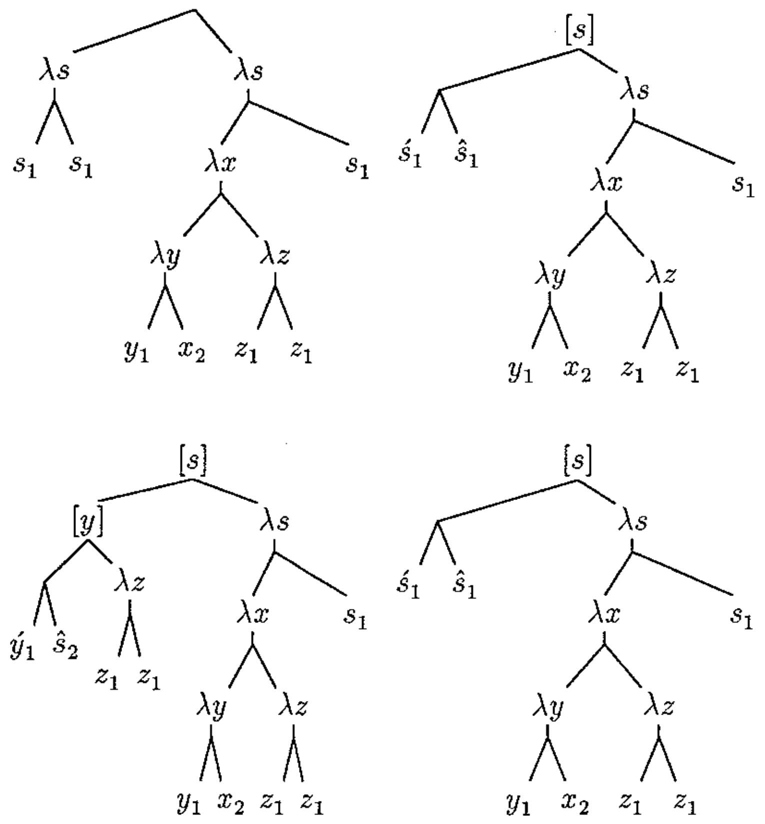


Figure 5.10: An example with unbounded last-instance relocations

amining real programs' sharing and execution behavior and studying their interaction with the set of techniques chosen for a particular implementation (Chapter 6 describes some extensions that would probably be used).

β-reduction. Once the redex is localized and in place, replacing variables (with conventional pointers in a λ_g-interpreter, and with λ_s-pointers in a λ_s-interpreter) is straightforward. A conventional graph reducer walks the graph looking for bound variables ($O(n \lg A)$ in the worst case); a λ_s-interpreter on an FFPM does it in $O(\lg A)$ time (Algorithm 5.8).

Tidying. If a λ_g-interpreter had to do tidying, it might include the removal of indirection nodes. Practical graph reducers avoid such things [165, pages 217–218], so I do not charge graph reduction for tidying.

The purpose of tidying in the λ_s-interpreter is to keep a λ_s-term in a form on which an FFPM implementation can use its associative matching effectively. Broadly speaking, we want to ensure that there is at most one s-connection to traverse between any two plain nodes.

After a β_s-reduction, any of the tidying rules may need to be applied. The purpose here is to ensure that each rule need not be applied more than once: since all the algorithms for tidying (Sections 5.2.4 and 5.2.5) take $O(\lg A)$ time, that will then give an $O(\lg A)$ time cost for the tidying phase as a whole.[11] One assumes, of course, that the λ_s-term was tidy before the reduction step began.

The [triv-ptee] rule replaces bound variables in the suspension's body with another variable; this cannot make any other rule applicable.

The removal of a useless suspension happens because it no longer has a bound variable. Such a removal cannot create the need for any additional tidying.

The [sus-rotl] rule re-orders suspensions; it cannot make another rule applicable. In particular, a "run" of [sus-rotl]'s (as in Figure 5.11) cannot happen, because it would require non-leftmost β_s-reductions to create the inner suspensions, which is not normal-order evaluation.

As discussed in Section 5.2.5, a given β_s-reduction may cause the need for an arbitrarily large number of applications of the [λ-up] rule. Fortunately, these always appear in the form $[^m\{^nB\}^n\ P_1]P_2]\ldots P_m]$ and the FFPM implementation of the [multi-λ-up] pseudo-rule can handle the whole thing in

[11]The annoying exception of the [multi-λ-up] pseudo-rule (Algorithm 5.14) will be dealt with shortly.

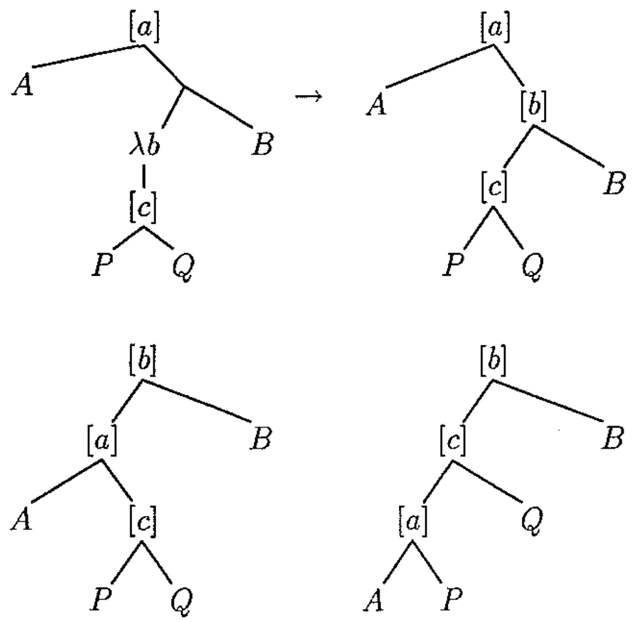


Figure 5.11: Impossible sequence of [sus-rotl]'s

one step.

Algorithm 5.14 takes $O(\lg A)$ time, except for the storage-management cost of making room for the n right braces to be inserted after the last right-bracket. In the utterly bizarre case of an n-node λ_s-term representing a function with $O(n)$ parameters, this storage-management time cost could be $O(n)$, wrecking the algorithm's overall worst-case time complexity. This $O(n)$ time cost need not be taken seriously, as it is a vestige of the particular linear representation I have used (chosen mainly for simplicity). In Table 5.2, I report the tidying cost for both the simple representation used here and for a "fancy" representation that would be likely used in practice.

The use of suspension lists and "clumped" λ-abstractions, explained in Section 6.1, is an alternate way to do many [λ-up]'s at once without resorting to the hackery of Algorithm 5.14 for the [multi-λ-up] pseudo-rule. Implementations of the rules for such a representation do not suffer the theoretical storage-management time costs just mentioned; the "fancy representation" column of Table 5.2 would apply.

This completes the explanation of Table 5.2, confirming that the time complexity of an FFPM implementation of a λ_s-interpreter is equal to or better than a GAM-machine implementation of a λ_g-interpreter with the exception of last-instance relocations.

Time for k reduction steps

An $O(t^2)$ bound, with $O(n)$ copying. The previous section confirms that the worst-case time complexities of a λ_g-interpreter and a λ_s-interpreter on a particular reduction step are nearly the same, with the latter perhaps suffering an $O(n)$ last-instance relocation. Can the extra cost of last-instance relocation over k reduction steps be bounded?[12]

Assume a sequence of k reductions, $T^1 \rightarrow T^2 \rightarrow \ldots \rightarrow T^k$. The size of term T^i is n_i; Section 5.3.1 showed that the sizes of the λ_g- and λ_s-term representations will differ by a constant factor, at most. We also know that the times for the non-last-instance-relocation parts of each reduction step in the two interpreters are asymptotically equivalent; call this time t; $t \geq Ck$ (for some constant C, to be ignored).

The *least* possible work to be done is constructing the largest term in memory; that is, $t \geq C\max(n_i)$, for some constant C (again ignored); t is the best possible graph-reduction time.

From Table 5.2, the *most* possible extra work in a λ_s-interpreter for k steps is k last-instance relocations of size $\max(n_i)$, which takes time of at most $k\max(n_i) \leq kt \leq t^2$. Therefore, the bound is

$$\begin{array}{rcl} \text{worst-case for } \lambda_s\text{-interpreter} & \leq & t + t^2 \\ & = & O(t^2). \end{array}$$

This is not a wonderful bound, but it confirms that the defeat of exponential blow-ups by non-naive tree reduction is not an order-notation accounting trick in Table 5.2.

An $O(t \lg t)$ bound, with $O(\lg n)$ copying. The analysis above presumes the usual $O(n)$ memory-copying time of an FFPM or a conventional GAM machine; it is worth mentioning the result's sensitivity to that assumption.

What if an n-node last-instance relocation could be done with a richer interconnection network in $O(\lg n)$ time? In this case, the extra work for last-instance relocations in k steps is at most $O(k\lg(\max(n_i)))$, which is bounded by $O(k\lg t) \leq O(t\lg t)$. Now the bound is

$$\begin{array}{rcl} \text{worst-case for } \lambda_s\text{-interpreter} & \leq & t + t\lg t \\ & = & O(t\lg t). \end{array}$$

The difference between the $O(t^2)$ and $O(t\lg t)$ bounds shows the effect of the modest data-movement network on an FFPM.

[12]I am completely indebted to David Plaisted for his help on this section.

An $O(t \lg t)$ bound, with $O(n)$ copying. It is likely that an $O(t \lg t)$ bound can be achieved even with $O(n)$ copying, if last-instance relocation is redefined to do *just enough* of the relocation to let the next reduction proceed. For example, instead of copying those whole λ_s-abstraction term, one could copy just its root node, then put in a special λ_s-pointer as the body. The new λ_s-pointer would be aimed at the uncopied abstraction-body, and more copying would be done later, if required. That would mean copying a constant number of nodes per relocation, at most. This idea does not fit squarely into the current mechanisms and the extra machinery (*not* worked out) is beyond the scope of my work here.

5.4 Previous FFP Machine implementations of the λ-calculus

Part of Backus's early work (1973) on functional programming included "λ-Red" (reduction) languages; they were "closed applicative languages which resemble the λ-calculus" [13]. They had variables and general substitution and worked by innermost evaluation. Since Magó was designing for Backus's languages, the initial drafts of Magó's original FFPM design (1979) supported λ-Red languages. The FFPM was a "λ-calculus machine" to begin with! Magó assumed pure tree-reduction-with-copying and made no efforts toward sharing at that time.

Dybvig's Ph.D. dissertation describes an implementation of Scheme (a statically-scoped LISP) for an FFPM [62]; he translates Scheme code into a specialized FFP language. The interpreter uses environments (represented as FFP sequences) and indexes into them with FFP selector functions. Dybvig's method copies these environments around unapologetically, though he uses special primitives to "trim" them of unneeded elements.

Plaisted [170; 171] looks at the ramifications of adding a richer interconnection network (a "$\Delta 2^i$ network") to an FFPM. He shows how this network leads to reasonable asymptotic complexity for parallel-innermost term rewriting.

Plaisted goes on to "give methods by which lazy evaluation and a version of graph reduction may be simulated fairly efficiently on the FFP Machine" [171, page 230]. He uses a notation comparable to marked and unmarked delimiters (Section 5.1.3). His "evaluation" parentheses, written as $(_e \cdots _e)$, are marked; unadorned parentheses are unmarked. No reduction takes place except inside an e-parenthesized term. Plaisted also uses "delay" parentheses to inhibit evaluation, including that of enclosed e-parenthesized terms. His equivalent of a suspension $[_x B\ P]$ is a **where**-expression:

$$[B \textbf{ where } \langle x\ P\rangle] \text{ or } [B \textbf{ where } \langle x_1\ P_1\rangle \ldots \langle x_n\ P_n\rangle].$$

The latter form is a full-blown environment.[13] Plaisted's equivalent of β-reduction creates a **where**-expression, as the β_s rule does with suspensions. The placement of e- and d-parentheses implements the desired evaluation order, using the FFPM trick of "colored parentheses."

Plaisted does not suggest rules to move **where**-expressions around as the λ_s-interpreter does with suspensions. When a variable x in B inside $(_e[B \textbf{ where } \langle x\ P\rangle]_e)$ needs to be filled in with P, he copies. This makes sense because his premise was adding a richer network for data movement to an FFPM. It certainly avoids the complications of following λ_s-pointers! He also uses the fast network to "collect" duplicate subexpressions at run-time: a term E with several occurrences of F becomes $[E' \textbf{ where } \langle x\ F\rangle]$, with the F's in E replaced by x, giving E'. This achieves a sharing of free expressions comparable to graph reduction.

Plaisted uses plain string-named variables, acknowledging binding indices as another alternative. He gives a matching algorithm for finding string-named bound variables (and useless **where**-expressions with no bound variables).

[13]I have altered Plaisted's notation slightly, so it blends with other parts of this dissertation.

Chapter 6

Embellishments

Fortran 8x appears to be well suited
to the functional style of programming.
— Page and Barasch (1985).

This chapter describes extensions that could make the λ_s-interpreter of Chapter 4 into a more practical basis for a parallel implementation of a lazy functional language. I assume FFP Machine (FFPM)-like target hardware that supports a linear program-representation and fast scan primitives, as in Chapter 5. This chapter includes much opinion and no data; the true test of any set of ideas is its impact on the performance of realistic functional programs.

All of the usual λ-calculus options would be available while designing a practical computational base: the different normal forms, the different evaluation orders, and so on. Peyton Jones's book about implementation describes many of these alternatives [165].

6.1 Suspension lists

A basic desideratum in parallel reduction machines is for large rewrite rules that do considerable useful work per step; there is a non-trivial overhead per step, at least in all machines designed so far. In an FFPM, this desideratum is reflected in the basic data type, dynamic arrays, and its preference for shallow structures with many large components (a deep binary tree with

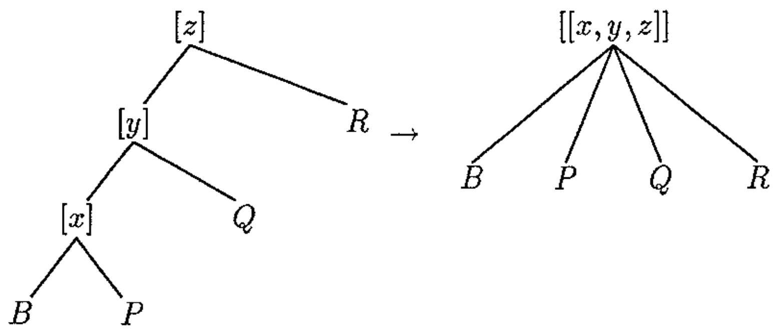


Figure 6.1: Left-skewed suspensions become a suspension list

measly integers at its leaves is an example of the opposite) [146]. Shallowness is important because an FFPM often moves up and down a tree structure by setting and unsetting activeness information on delimiters; this can be done at most once per cycle. (The interpreter of Chapter 5 uses a "global checking" phase to avoid some of this re-setting, but such a method might not always be applicable or desirable.)

The rules of the λ_s-interpreter in Chapter 4, notably the [sus-rotl] rule, conspire to create unbalanced, left-linear λ_s-trees of suspensions. This structure may be replaced with a *suspension list*; Figure 6.1 shows an example without and with a suspension list ("[[ ]]" in the tree). The "meaning" of a suspension list is that of its left-linear, unraveled equivalent λ_s-term. The names x, y, z in $[[x, y, z]]$ are decorative of course, and match the pointees in the obvious left-to-right way.

A suspension list has a body subterm and n pointee subterms, $n \geq 1$. In counting binders (to determine binding indices), such a suspension list counts as n. There is a difference between binding indices in a suspension-list term and its ordinary λ_s-term equivalent. Figure 6.2 gives an example.

The rules of the λ_s-interpreter must be adjusted for suspension lists. Because they are more like FFPM dynamic arrays than simple suspensions, one would hope the rules' actions would be larger, and that this benefit would outweigh the increased tedium of managing the suspension lists.

Incrementing and decrementing of binding indices must account for the multi-binder nature of suspension lists (easy). The [triv-body] rule, which is a "keep-it-or-throw-it-away" binary selector in its ordinary λ_s-term form, becomes a proper selector, choosing one (or none) of the suspension list's pointees.

The [sus-rotl] rule goes away and is replaced by a rule that subsumes

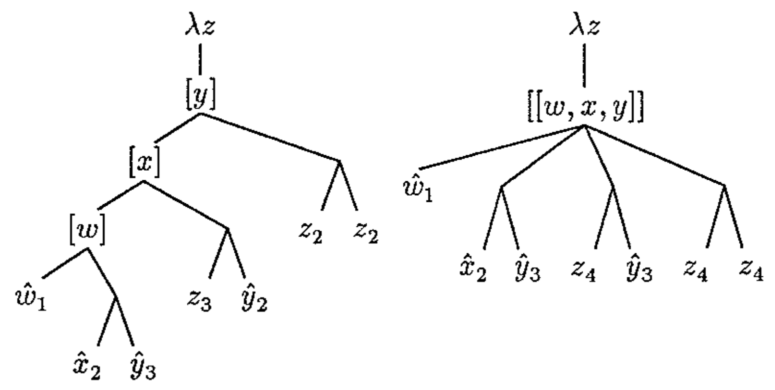


Figure 6.2: Binding indices change in a suspension list

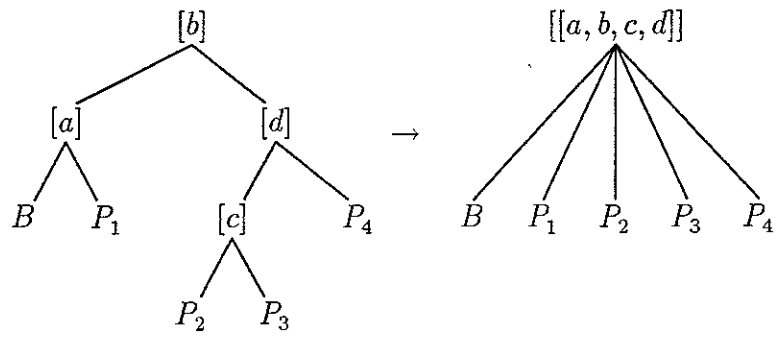


Figure 6.3: Suspension lists collapse into their parent list

suspension lists into their parent list. Ensuring that every pointee has a plain node at its root is still the goal. Figure 6.3 gives an example of [sus-rotl]-like suspension-list collapsing.

Lumping suspensions into suspension lists is similar to a standard practice with λ-abstractions in a name-free calculus: representing $\lambda a.\{\lambda b.\{\lambda c.\{B\}\}\}$ by $\lambda^3.\{B\}$. $\lambda^n.\{T\}$ nodes count as n binders. This representation would also tend to make flatter λ_s-trees and is probably a good idea. A rule to subsume subordinate λ-abstractions would be needed: $\lambda^m.\{\lambda^n.\{B\}\} \rightarrow \lambda^{m+n}.\{B\}$.

If one is going to be a truly enthusiastic flattener, perhaps the λ-applications should also be replaced with *λ-application lists*. A left-linear tree of λ-applications, as in Figure 6.4, would be replaced by a λ-application list (shown by "(( ))" in the tree). A λ-application list may only absorb subordinate λ-applications on its left end, a greater-than-usual constraint.

The beauty of all this listifying is that applying a function to its several

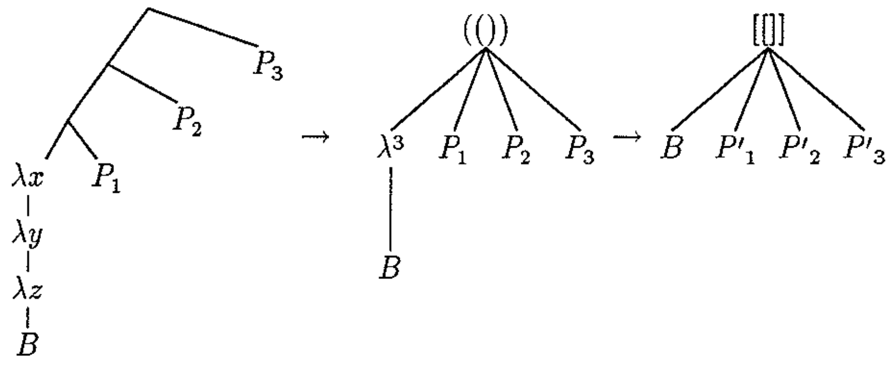


Figure 6.4: λ-application list and subsequent reduction

arguments is now a one-step operation, as Figure 6.4 shows. Optimizations of this operation are common in the literature. Paul Watson uses "multiple β-substitution" in his thesis, for multi-argument substitutions in which the partial evaluations are not shared [206]. Berkling goes further in his 1986 paper on head order reduction; he uses η-expansion to make the multi-argument redexes as large as possible, calling the operation that follows "β-reduction-in-the-large" [25]. The exact suspension- and λ-application-list analogues of his operations would be worth figuring out. Though multi-argument reduction may turn out to be impractical—Hartel and Veen report that for "four medium-sized programs at least 90 per cent of all functions have one or two arguments" [89, page 247]—it suggests a desirable direction, one that might be well served by an aggressive compiler.

As part of his λ-calculus work, Révész has proposed a language extension in which lists are fully integrated into the calculus [176, Chapter 4]. An example of a term might be $\lambda x.\{\langle E_1, E_2, \ldots, E_n\rangle\}$. The syntax of this extension and the implications thereof are too involved to describe here; however, I think such a base calculus might be appropriate for the style of reduction being promoted here.

6.2 Dealing with recursion

Recursive functions are pervasive in lazy functional programming. One reason for graph reduction is that cycles in the program graph can represent recursion.

With ordinary tree reduction, there is no way for a λ-term to refer to itself. With suspensions, it *is* possible, by relaxing the restriction that a

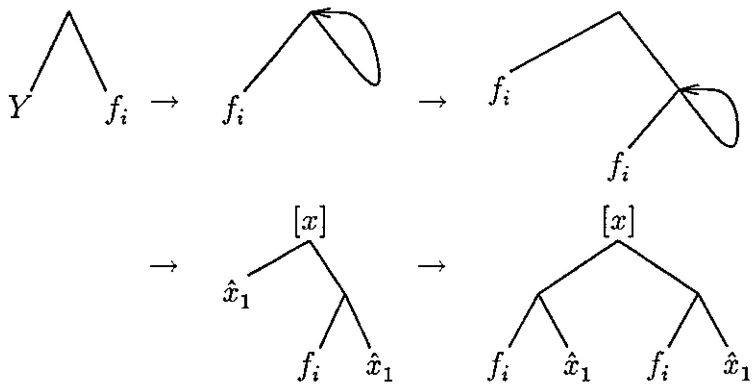


Figure 6.5: Two Y combinator reductions, graphs and suspensions

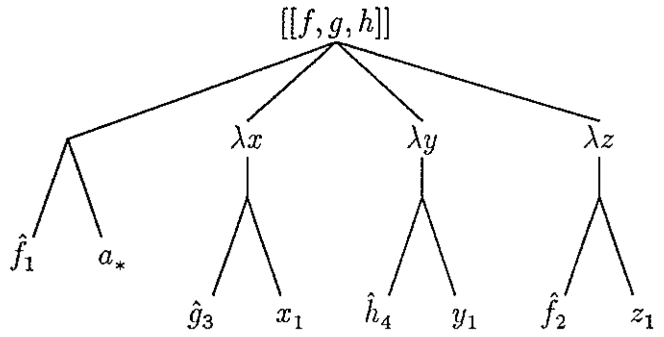


Figure 6.6: Mutually recursive functions in a suspension list

suspension cannot have a bound variable in its pointee. Figure 6.5 shows two reductions of the Y combinator; the cyclic-graph reductions are shown above, the bound-variable-in-pointee reduction below. The [triv-body] rule must be changed to check for bound variables in the pointee; the rule must *not* be applied to the λ_s-term in Figure 6.5.

The bound-variable-in-pointee method may not work well for sets of mutually recursive functions. Such functions can, however, be handled by extending the idea to suspension lists. Figure 6.6 shows three mutually recursive functions f, g, and h tangled up in a suspension list.

6.3 Exotic suspensions

If one thinks about suspensions in the λ_s-interpreter as active entities, their basic operations are to scan for bound variables that need filling, then check if the pointee is appropriate for copying (i.e., it is a λ-abstraction), then do the copying.

One can imagine "smarter" suspensions, probably working in concert with other machine primitives. For example, if the machine had an array-of-numbers primitive, instead of copying the array so a selector could proceed, the smart suspension would do the selection itself and simply copy the result.

6.4 More parallelism

To have a practical fine-grained parallel machine, there must be *plenty* of parallelism. Normal-order evaluation of the pure λ-calculus is essentially serial and offers little parallelism. A richer base language with more primitives allows more parallelism, and a less stringent evaluation order might help, too. For example, strict functions' arguments may be evaluated eagerly (and concurrently); strictness analysis may find other functions with similar properties. Using suspensions does not preclude these standard techniques.

The β_s rule and the tidying rules are fast and do not consume space (assuming an implementation like the one in Chapter 5); the rules may be applied in as many places as possible at once. If the expensive operations—notably filling FollowFill pointers—are still applied only when normal-order evaluation demands it, then its termination properties will be preserved.

6.5 Ordering in λ_s-terms and supercombinators

Section 4.5.6 discussed how suspensions' movement or reordering in a λ_s-tree is constrained by binders remaining visible to their bound variables. Suspension lists with bound variables in pointees allow greater freedom in this ordering. This might benefit a particular implementation.

An extreme example of re-ordering freedom comes from converting a λ-term to supercombinator form; Figure 6.7 shows a modified example from Peyton Jones's book [165, page 226]. The free variable x_2 is abstracted from the term $\lambda y.\{(y_1\ x_2)\}$ and the two resulting supercombinators $\$X$ and $\$Y$ are put in a top-level suspension list. The order in which they appear as pointees in the list is completely arbitrary (it could be $[[\$X, \$Y]]$). It is worth mentioning that *any* λ-term converted to supercombinators would be of this form: a single top-level suspension list with trees of λ-applications

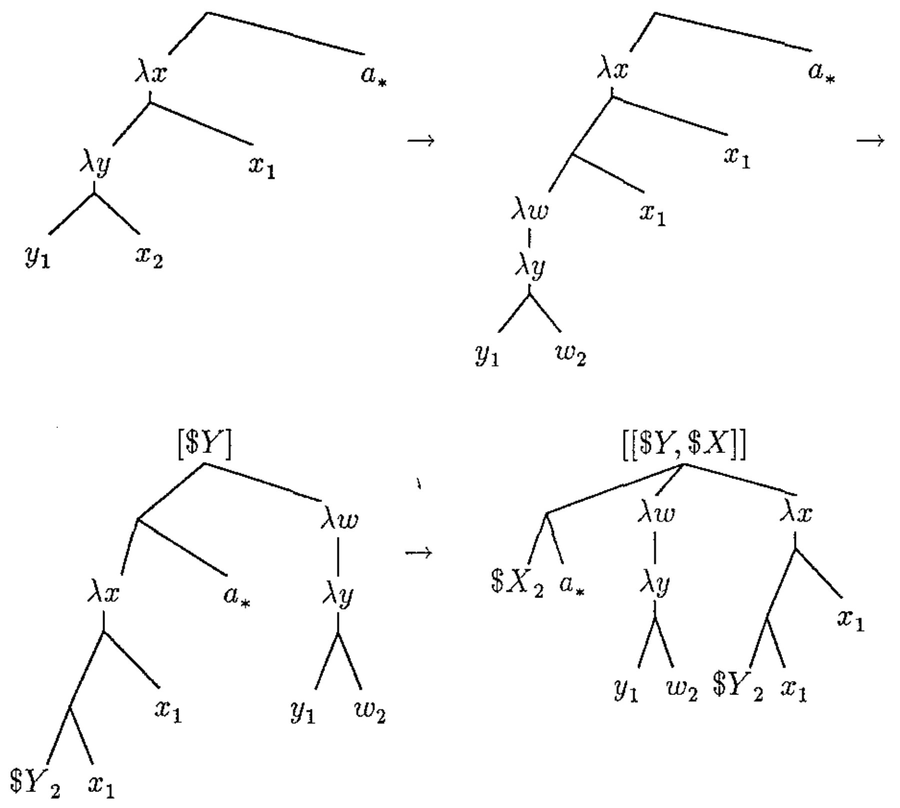


Figure 6.7: Transforming to supercombinators, with a suspension list

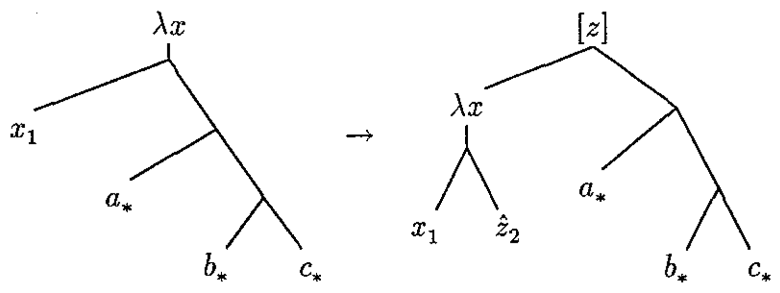


Figure 6.8: Abstracting a free expression

and variables (no λ-abstractions) dangling from it.

A comparable technique works for abstracting free expressions; Figure 6.8 shows the term $(a\ (b\ c))$ lifted out of the abstraction $\lambda x.\{(x\ (a\ (b\ c)))\}$. Preprocessing λ_s-terms in this way is necessary for a λ_s-interpreter to be fully lazy, as Arvind et al. showed [9]. In general, most techniques for variable-abstraction, λ-lifting, etc., seem to carry over quite directly to the suspension-based approach.

6.6 Speculative copying

Asymptotic properties aside, I think it is clear that heavy use of suspensions and λ_s-pointer-following are not an easy ticket to high-speed computation. Suspensions should only be used in cases where they really help.

All copying beyond the necessary is speculative and potentially wasted. However, a parallel-machine designer welcomes speculative copying that improves locality or parallelism at a modest cost. There is some chance that decent heuristics to guide such copying would emerge for real programs. An example heuristic: "λ-abstractions are usually small: copy them." Or, assuming array primitives: "Never copy arrays." This general approach is consistent with the common string/tree reduction-machine approach of overcoming a lack of sharing by specialized support for more complex data structures (Section 5.1.11).

More subtle variations of speculative copying spring to mind. For example, last-instance relocation of a suspension's pointee is a dead loss compared to graph reduction. This problem might almost always be alleviated by copying earlier than necessary: when doing the next-but-last copy of the pointee, one could overwrite the last bound variable as well (and delete the useless suspension). Alternately, one could make as many copies as possible within some "budget" for new cells allocated. (One would presumably fill the leftmost bound variables on the hunch that they would be the next ones needed.) Scan primitives readily support counting bound-variables, sizing the pointee, and marking some of variables for overwriting after a local calculation of the number of new cells expected to a cell's left.

Copying costs are mightily affected by an architecture's interconnection scheme. A richer network or a topology that favors a particular style of copying would shift the balance about what speculative copying is worth doing.

6.7 Going further with binding indices

The FFPM implementation of the λ_s-interpreter in Chapter 5 does λ_s-pointer following and filling by looking for Follow*Fill-marked binding indices and matching simple two-cell patterns. Here, I want to mention an alternate technique that is interesting in its own right.

In partitioning, an FFPM gives hardware resources to innermost active λ_s-terms and lets them proceed (Section 5.1.3). Pre-order walking of λ_s-terms to find redexes can be implemented by marking subterms as active (corresponding to a recursive call in a tree-walker) and having those subterms unmark themselves when finished (corresponding to a return from the recursive call). Because the following of λ_s-pointers introduces non-local jumping around the tree, repeated markings and unmarkings of "activeness" would make for a poor implementation of the λ_s-interpreter (even those the ML version onestepS (page 72) works precisely by doing all those recursive calls to move up and down the λ_s-term). This is the reason for the "global checking" phase in the implementation in Chapter 5.

I found a fast FFPM-style way of doing the "returns" back from a suspension down to the λ_s-pointer that called it. The gimmick is to give "binding indices" to suspensions and λ-applications as well as variables. These may then be used beneficially. Consider Figure 6.9a: all the nodes between the variable $\acute{x}_2$ and its suspension $[x]$ have binding indices pointing to that suspension. The top suspension has a "binding index" of 0, marking it as active. If that suspension then does a simple "compare application-level (RAL) with binding-index" operation (there were plenty of those in Chapter 5), then all its bound suspensions and λ-applications can be detected and marked active with binding-index 0. The configuration of "activeness" that existed before following the λ_s-pointer $\acute{x}_2$ has been restored! Figure 6.9b shows the result

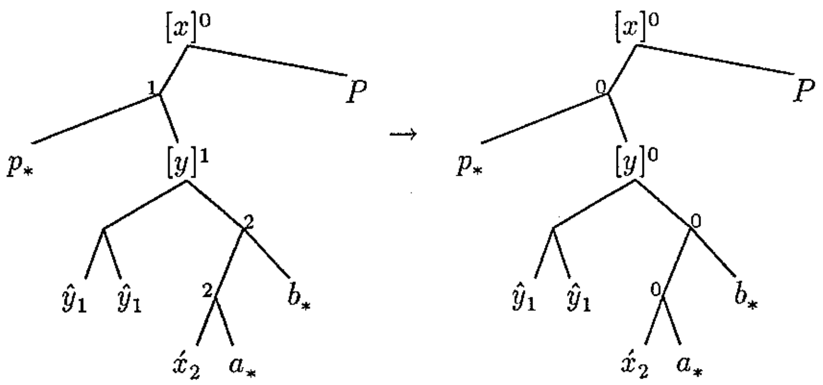


Figure 6.9: Binding indices on suspensions and λ-applications

of such an operation.

Some standard λ-lifting-style operations also work by giving binding indices to non-variable constructions (see Peyton Jones's book, again [165, pages 230 and 258]).

Chapter 7

Conclusions

I endeavour to give satisfaction, sir.
— C. Northcote Parkinson, *Jeeves* (1979).

The basic results are those I claimed in the thesis statement (Section 1.2).

- A suspension-based λ_s-interpreter (Chapter 4) is a correct implementation of the pure λ-calculus because its manipulations of λ_s-terms are isomorphic to the manipulations of λ_g-graphs by a graph-reducing λ_g-interpreter (Chapter 3). Theorem 4.9 (page 92) proves this λ_{sg}-equivalence.

- When the λ_s-interpreter is implemented on an FFP Machine (FFPM) or similar architecture, its worst-case space complexity is within a constant factor of that of a lazy-copying graph-reducer on a global-addressable-memory (GAM) machine. Please see Theorem 5.1.

- The worst-case time complexity of the FFPM interpreter is equal to or better than that of the GAM interpreter, except for the last-instance relocations of suspension pointees. Please see Table 5.2.

I conclude that graph reduction does *not* have an inherent advantage as a computational model to support lazy functional programming. I now review the major issues raised by comparing the two interpreters.

Reduction to β-normal form. Graph reduction that uses lazy copying is only suited to reduction to weak β-normal form (WBNF); it cannot cope with free variables in redexes. Also, binding indices cannot be used with this kind of graph reduction (Section 3.4.1). A graph reducer must either support α-conversion or use backpointers to avoid name-capture problems. Also, to enjoy maximal sharing with graph reduction, one must include expensive detection of maximal free expressions (MFEs) in each reduction step ("fully lazy" copying); when reducing to β-normal form (BNF), the less onerous "lazy" copying does *not* work (Section 3.3).

Suspension-based reduction to BNF works with binding indices, and lazy copying is a completely natural mechanism. If an implementer wants to use a reduction order or normal form that allows free variables in redexes—e.g., innermost spine reduction (see Peyton Jones [165, page 199]) or BNF—then suspension-based reduction is available.

Oddly, the FFPM implementation that does so well with suspension-based reduction to BNF finds reduction to WBNF *more* costly. Algorithms for finding out if a term is inside a λ-abstraction are ill-matched to what the hardware can do well.

Linear representation. The attractions of suspension-based reduction to computer architects center around a linear representation of program symbols. A symbol need *not* be globally addressable nor stored in a global resource.

Binding indices, used to avoid name-capture problems, are very amenable to manipulation with fast scan primitives. Algorithm 5.1, which detects all bound variables in a λ_s-term in $O(\lg n)$ time, is a beautiful example.

The use of associative matching to detect redexes and other "interesting" patterns of symbols is noteworthy. This matching can find a redex anywhere in a λ-term is as little as one step, whereas a graph reducer must necessarily chain through pointers to get there. An important aspect of this matching (and the other FFPM algorithms) is the modest amount of "parse-tree" information that must be synthesized from the raw symbols—no more than two selectors are ever needed.

I think it worthwhile to have presented a sizable example using the low-level techniques possible on an FFPM. I believe that these techniques would work just as well on *any* parallel architecture with fast scans and a locality-preserving linear program-representation.

The cost of a linear program-representation (besides the implementation cost) is "last-instance relocation," which means that the last copy of a λ_s-abstraction must be moved into place; these movements are gruesomely dissected beginning on page 145. I think the analysis there shows how constrained the "extra" copying of tree reduction can be.

The next step. I hope I have laid to rest the notion that "string" reduction is inherently, wildly inefficient for normal-order reduction. The basic question that follows is: Can the λ_s-interpreter of Chapter 4 be "grown" into a practical mechanism for the efficient execution of lazy functional programs? Intimately related to this question are the questions of what realistic functional programs actually do ("what happens above") and the constraints and properties of the target hardware architecture ("what happens below"). The truly successful architect for a parallel reduction machine will be master of all of these levels.

I would seriously consider using some of the tricks of Chapter 6. Suspension lists seem a clear winner (Section 6.1), and Révész's extensions to integrate lists directly into the calculus are no less intriguing (mentioned in the same section). I would allow bound-variables in suspension pointees and use that to implement recursion. I would do some speculative copying based on simple heuristics derived from real programs; an example might be, "if filling a suspension's next-but-last bound variable, fill the last one as well."

Graph reduction without pointers. Wadsworth's invention of graph reduction was a breakthrough for normal-order evaluation of the λ-calculus; it made the unthinkable thinkable. Subsequent development of sequential implementations (e.g., the G-machine) have removed the glaring weaknesses of graph reduction, so that it now forms the basis for quite-practical lazy functional programming systems.

Good parallel implementations of graph reduction seem less assured. They must inescapably contend with autonomous processors vying for a shared resource, the program λ_g-graph. The nature of the graphs does not build one's hopes for abundant locality. Yes, there are tricks, but ... Why not a suspension-based reduction mechanism that does the same reductions as graph reduction on each reduction step, and that has *exactly* the same sharing properties (assuming lazy copying)? Terms are represented by trees, no concept of global store need intrude, and no way to *address* the global store—pointers—need be supplied. Besides, suspension-based reduction works perfectly well even if redexes include free variables. Why not the benefits of graph reduction *without* pointers?

Appendix A

Programming with ML

This appendix introduces a small subset of Standard ML so you can read my programming examples. Common utility routines used in the dissertation are explained in Section A.2.

Wikströom's text [211] is a proper introduction to ML programming. There is much, much more to ML than I describe here.

A.1 The one-minute ML programmer

Data structures. Primitive data types include booleans true and false (type bool), character strings (type string), and integers (type int). Negative integers are written as ~1, ~2, ~3, ... The null type is unit.

I use only one compound data type, *tuples*. For example, (1, bool, 6) is a 3-tuple; its type is int * bool * int.

A **datatype** declaration introduces a user-defined type; for example,

datatype Tree = Leaf **of** int | Node **of** Tree * Tree.

Leaf and Node are *constructor* functions; their types are int → Tree and Tree * Tree → Tree, respectively. (ML is strongly-typed, and it is common practice to give a function's type along with its name.) My main **datatypes** are Term, for λ-terms with suspensions, and Gnode, for the nodes of a λ_g-graph.

References. ML allows pointers, or "references." A pointer to an int has type int ref. I use references in the graph-twiddling code (lots of Gnode refs).

For a bool ref x, x := true is an assignment, and !x is the value of whatever x points to (!x is x "dereferenced").

Functions. An ML function is usually written as a set of clauses, each specifying a *pattern* that an argument must match. For example:

```
fun leafcount (Leaf(x))      = 1
   | leafcount (Node(L,R)) = (leafcount L) + (leafcount R)
```

The function leafcount : Tree $\rightarrow$ int is defined with two clauses. The first says what to do with a Leaf and the second with a Node. Using patterns to unravel data structures is very common practice in ML.

An underline "_" in a pattern is a "don't care" variable, matching anything in that position.

The same result can be had with a **case** clause with patterns:

```
fun casecount T =
    (case T
     of Leaf(x) => 1
       | Node(L,R) => (leafcount L) + (leafcount R)
    )
```

A function doubletree : Tree $\rightarrow$ Tree, which doubles every leaf in the input Tree, giving a new Tree, might be:

```
fun doubletree (Leaf(n))      = Leaf(2 * n)
   | doubletree (Node(L,R)) = Node(doubletree L, doubletree R)
```

A value or function may be "cached" using a **let**...**in**...**end** expression; for example,

$$\textbf{let val } x = 40 + 2 \textbf{ in } x + x + x + x \textbf{ end}$$

is an expression with value $42 \times 4 = 168$. A doubletree variant might be:

```
fun doubletree' (Leaf(n)) =
    Leaf(2 * n)
  | doubletree' (Node(L,R)) =
    let val L' = doubletree' L
        val R' = doubletree' R
    in Node(L', R') end
```

Let expressions can also introduce local function definitions, as in this silly example:

```
fun add_two_to_tree (Leaf(n)) =
    let fun add_two_to x = x + 2
    in Leaf(add_two_to n) end
  | add_two_to_tree (Node(L,R)) =
    Node(add_two_to_tree L, add_two_to_tree R)
```

Function application associates to the left; parentheses override the implicit order. So, doubletree L, doubletree(L), and (doubletree L) are all the same.

Polymorphic functions. In ML, one may define *polymorphic* functions that accept arguments of more than one type. For example,

```
fun first_of_tuple (x, y) = x
```

accepts 2-tuples with elements of any type and returns the first; it has type 'a * 'b → 'a. Just watch out for types written as 'a, 'b, 'c, ...

Higher-order functions. Higher-order functions are those that take functions as arguments or that return functions as results. The higher-order functions in this dissertation are partial applications of curried functions. (If that sounds *too* confusing, you may wish to consult a functional programming text; however, the examples here may get you through.)

```
fun add' (x, y) = (x + y):int (* coerce; '+' is overloaded *)
fun add x y = (x + y):int
```

The functions shown are add' : (int * int) → int and add : int → int → int. If invoked, e.g., (add' (3, 9)) or (add 3 9), both give the same answer. The function add, however, may be *partially* applied (with fewer than its full two arguments), as in (add 3), in which case it returns the *function* that adds three to its argument. The function (add 3) has type int → int; when it is applied to 9, we get the expected answer 12 (type int).

Besides add, a curried version of the built-in +, I also use orElse : bool → bool → bool, a curried version of the infix **orelse**.

This style of partially applying functions to yield other functions is important only for the functions chk_vars (page 170) and mod_vars (page 171).

```
(* Predicate functions is_app : Term → bool  (a λ_s-application?), is_lam : Term → bool
   (a λ_s-abstraction?), and is_ptr : Term → bool  (a λ_s-pointer?).
*)
fun is_app (App(_,_)) = true
  | is_app other      = false

fun is_lam (Lam(_,_)) = true
  | is_lam other      = false

fun is_ptr (Var(_,Ptr,_))          = true
  | is_ptr (Var(_,FollowFill,_))   = true
  | is_ptr (Var(_,FollowNoFill,_)) = true
  | is_ptr (Var(_,Followed,_))     = true
  | is_ptr other                   = false
```

Exceptions. ML allows elaborate exception-handling. For my purposes, however, it is simple: if you see "**raise** something_bad_wrong," it is a fatal, not-supposed-to-happen error.

A.2 Utility functions

A.2.1 λ_s-term functions

This section presents the ML functions that support the code for the λ_s-interpreter in Chapter 4. The functions chk_vars (page 170), mod_vars (page 171), and subst (page 172) are important, the rest are just necessary labor.

```
(* Function
   chk_vars : (int→int→int*VarMark*string→'a)→('a→'a→'a)→'b→int→int→Term → 'a
   .
   (A rather horrible-looking type, no?) Many operations on Terms are of the form "Go
   down and look at all the variables (binding indices, really), check for some condition,
   and accumulate the results." For example, given the query, "Does this suspension
   have a bound variable?", we check each variable's binding index against the nesting
   level (which must be accumulated); if equal, true else false. We orelse together all
   the variable-answers and that is the answer.
   mod_vars (page 171) is a similar function, except it modified Vars (producing new
   Terms), rather than just checking them.
   If instead all the leaves report '1' and the "connective" is addition, then we have a
   leaf counter. I have given it as an example below. Now, the non-obvious arguments
   to chk_vars:

   levl and levh: Accumulate "low" and "high" numbers defining the range of nesting
          levels of interest. As we move down the tree, these numbers are bumped up
          when we meet a binding site (Lam or Sus). Often, levl and levh are the same
          ... or perhaps very far apart (e.g., 2 and MAXVAR).

   check: The function applied to Vars to produce the values to accumulate. It is really
          applied to the levl and levh level-numbers and the "guts" of a Var, hence the
          exotic type int → int → int * VarMark * string → 'a.

   connect and unit: The function that accumulates the values from the Vars; unit is
          the "unit value" used to get things going. Common combinations would be
          add,0 or orElse,false.
*)
fun chk_vars check connect unit levl levh (App(M, N)) =
     connect (chk_vars check connect unit levl levh M)
             (chk_vars check connect unit levl levh N)
  | chk_vars check connect unit levl levh (Sus(B, P, _)) =
    let val nlevl = levl + 1
        val nlevh = levh + 1
    in connect (chk_vars check connect unit nlevl nlevh B)
               (chk_vars check connect unit nlevl nlevh P) end
  | chk_vars check connect unit levl levh (Lam(B, _)) =
        (chk_vars check connect unit (levl+1) (levh+1) B)
  | chk_vars check connect unit levl levh (Var(bi,vmk,n)) =
        (check levl levh (bi,vmk,n))

fun leafcount T = (* a chk_vars example *)
let fun var_counts_one levl levh (_,_,_) = 1
in chk_vars var_counts_one add 0 1 1 (* levels don't matter *) T end
```

```
(* Function mod_vars :
   (int→int→int*VarMark*string→bool)→
   (int→int→int*VarMark*string→int*VarMark*string)→
   int→int→Term→Term .
   This is a sister function to chk_vars, except that it modifies the variables and returns a
   new Term, rather than just checking the variables and accumulating the information.
   Please see the documentation for chk_vars (page 170).
*)
fun mod_vars sel modfn levl levh (App(M, N)) =
     App(mod_vars sel modfn levl levh M, mod_vars sel modfn levl levh N)
   | mod_vars sel modfn levl levh (Sus(M, N, n)) =
     let val nlevl = levl + 1
         val nlevh = levh + 1
     in
         Sus(mod_vars sel modfn nlevl nlevh M, mod_vars sel modfn nlevl nlevh N, n)
     end
   | mod_vars sel modfn levl levh (Lam(B, n)) =
     Lam(mod_vars sel modfn (levl+1) (levh+1) B, n)
   | mod_vars sel modfn levl levh (Var(bi, vmk, n)) =
     if (sel levl levh (bi, vmk, n)) then
         Var(modfn levl levh (bi, vmk, n))
     else
         Var(bi, vmk, n)

(* The most common use of mod_vars is to change variables' binding indices. The
   function incr_var_range  adjusts all the binding indices in a range of nesting levels
   (low to high) by an incr number.
   The functions incr_free_vars1, incr_free_vars2, and incr_bd_vars are just convenient ways
   to call incr_var_range. The two versions of incr_free_vars are needed because the binding
   indices for free variables start at '1' or '2', depending on where you start counting.
*)
fun incr_var_range low high incr T =
let
    fun in_range (levl:int) (levh:int) (bi, vmk, n) = (bi ≥ levl andalso bi ≤ levh)
    fun incr_var _ _ ((bi:int), vmk, n)             = ((bi+incr),vmk, n)
in mod_vars in_range incr_var low high T end

fun incr_free_vars1 incr T = incr_var_range 1 MAXVAR incr T

fun incr_free_vars2 incr T = incr_var_range 2 MAXVAR incr T

fun incr_bd_vars incr T = incr_var_range 1 1 incr T
(* Levels 1 1 assume we are working inside a Term *)
```

```
(*
  These functions are tests for various flavors of "boundness" that work with chk_vars
  (page 170) and mod_vars (page 171). The only hard one is is_higher_up_follow_ptr,
  which checks if a Follow λs-pointer's binding index says that its binding site is above
  the point indicated by the nesting level.
*)
fun is_bd_var_or_ptr levl levh ((bi:int),vmk,n) = (bi ≥ levl) andalso (bi ≤ levh)

fun is_bd_follow_ptr levl _ (bi,FollowFill,_)     = (levl = bi)
  | is_bd_follow_ptr levl _ (bi,FollowNoFill,_) = (levl = bi)
  | is_bd_follow_ptr levl _ (_,_,_)              = false

fun is_bd_follow_fill_ptr levl _ (bi,FollowFill,_) = (levl = bi)
  | is_bd_follow_fill_ptr levl _ (_,_,_)           = false

fun is_higher_up_follow_ptr (levl:int) _ (bi,FollowFill,_) = (levl < bi)
  | is_higher_up_follow_ptr levl _ (bi,FollowNoFill,_)    = (levl < bi)
  | is_higher_up_follow_ptr levl _ (_,_,_)                = false
```

```
(* subst : (int → int → int * VarMark * string → bool) → int → int → Term → Term →
   Term
   This function substitutes term S into term T for all Variables selected by the
   function sel. Uses incr_free_vars1 (page 171)—this is because, as S is "dragged down"
   the tree T, its free variables are getting further away from their binding sites.
   subst is in much the same spirit as chk_vars (page 170) or mod_vars (page 171).
   std_subst is simply a convenient way to call the more general subst.
*)
fun subst sel levl levh S (App(M, N)) =
    App((subst sel levl levh S M), (subst sel levl levh S N))
  | subst sel levl levh S (Lam(B, n)) =
    Lam((subst sel (levl+1) (levh+1) (incr_free_vars1 1 S) B), n)
  | subst sel levl levh S (Var(bi,vmk,n)) =
    if (sel levl levh (bi,vmk,n)) then S else Var(bi,vmk,n)
  | subst sel levl levh S (Sus(B, P, n)) =
    let val nlevl  = levl + 1
        val nlevh  = levh + 1
        val S'     = (incr_free_vars1 1 S)
        val B'     = (subst sel nlevl nlevh S' B)
        val P'     = (subst sel nlevl nlevh S' P)
    in Sus(B', P', n) end
fun std_subst S T = (* the most common use of subst *)
  subst is_bd_var_or_ptr 1 1 S T
```

```
(* swap_levs : Term → Term.
   A highly miscellaneous support function for tidyterm (page 88) that increments bind-
   ing indices on level ilev by 1 and decrements binding indices on level dlev by 1.
*)
fun swap_levs ilev dlev (App(M,N)) =
     App(swap_levs ilev dlev M, swap_levs ilev dlev N)
   | swap_levs ilev dlev (Sus(B,P,n)) =
     Sus(swap_levs (ilev+1) (dlev+1) B, swap_levs (ilev+1) (dlev+1) P, n)
   | swap_levs ilev dlev (Lam(B, n)) =
     Lam(swap_levs (ilev+1) (dlev+1) B, n)
   | swap_levs ilev dlev (Var(bi,vmk,n)) =
     if bi = ilev then
        Var(bi+1,vmk,n)
     else if bi = dlev then
        Var(bi-1,vmk,n)
     else
        Var(bi,vmk,n)
(* Uses mod_vars (page 171) to turn bound plain-variables into Ptr variables. *)
fun ptrize_bd_vars T =
let
     fun make_ptr lev _ (bi,NotPtr,n) = (bi, Ptr, n)
        | make_ptr lev _ ( _, _,_)       = raise ptrize_bd_vars_error
in mod_vars is_bd_var_or_ptr make_ptr 0 0 T end
```

A.2.2 λ_g-graph functions

This section includes the few extra functions needed to support the λ_g-interpreter in Chapter 3. Though these functions are hideous-looking because of all the pattern-matching on graph structure, they are all quite simple.

```
(* Simple functions to change reference counts and set subbed bits. *)
fun incr_refcnt bump (G as ref (AppG(_,_,_,(_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump
  | incr_refcnt bump (G as ref (LamG(_,_,_,(_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump
  | incr_refcnt bump (G as ref (VarG(_,_, (_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump

and set_subbed bval (ref (AppG(M,N,_,(subbed,_,_,_)))) = (subbed := bval)
  | set_subbed bval (ref (LamG(B,_,_,(subbed,_,_,_))))   = (subbed := bval)
  | set_subbed bval (ref (VarG(_,_, (subbed,_,_,_))))    = (subbed := bval)
(* . Set visited bits to bval. *)
and mk_graph_visited bval (ref (AppG(M,N,_,(_,_,visited,_)))) =
     (visited := bval; mk_graph_visited bval M; mk_graph_visited bval N)
  | mk_graph_visited bval (ref (LamG(B,_,_,(_,_,visited,_)))) =
     (visited := bval; mk_graph_visited bval B)
  | mk_graph_visited bval (ref (VarG(_,_, (_,_,visited,_)))) =
     (visited := bval)
```

```
(* Simple functions to change reference counts and set subbed bits. *)
fun incr_refcnt bump (G as ref (AppG(_,_,_,(_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump
  | incr_refcnt bump (G as ref (LamG(_,_,_,(_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump
  | incr_refcnt bump (G as ref (VarG(_,_, (_,refcnt,_,_)))) =
      refcnt := (!refcnt) + bump

and set_subbed bval (ref (AppG(M,N,_,(subbed,_,_,_)))) = (subbed := bval)
  | set_subbed bval (ref (LamG(B,_,_,(subbed,_,_,_))))   = (subbed := bval)
  | set_subbed bval (ref (VarG(_,_, (subbed,_,_,_))))    = (subbed := bval)
(* . Set visited bits to bval. *)
and mk_graph_visited bval (ref (AppG(M,N,_,(_,_,visited,_)))) =
     (visited := bval; mk_graph_visited bval M; mk_graph_visited bval N)
  | mk_graph_visited bval (ref (LamG(B,_,_,(_,_,visited,_)))) =
     (visited := bval; mk_graph_visited bval B)
  | mk_graph_visited bval (ref (VarG(_,_, (_,_,visited,_)))) =
     (visited := bval)
```

```
(* The function rm_indir_nodes : Gnode ref → Gnode ref  removes any AppG nodes that
   have become indirection nodes. It turns off all the visited bits (with mk_graph_visited
   (page 175)) then uses the local function rm2 to do the work; rm2 sets visited bits and
   does not do any re-visiting. Reference counts are adjusted with incr_refcnt (page 175).
*)
and rm_indir_nodes (ref (AppG(M,N,(ref true),(_,_,_,_)))) =
    (* a toplevel indirection node *)
    (rm_indir_nodes M)
  | rm_indir_nodes ptr =
    let fun rm2 (G as ref (AppG(M,N,(ref true),(_,refcnt,visited,_)))) =
          (* this is an indirection node; repeat visits; fiddle refcounts *)
          let val M' = (rm2 M)
          in ((incr_refcnt ((!refcnt) - 1) M'); M') end
        | rm2 (app_ptr as ref (AppG(M,N,(ref false),(_,_,visited as (ref false),_)))) =
          let val _ = (visited := true)
              val M' = (rm2 M)
              val N' = (rm2 N)
          in M := !M'; N := !N'; app_ptr end
        | rm2 (lam_ptr as ref (LamG(B,n,_,(_,_,visited as (ref false),_)))) =
          let val _ = (visited := true)
              val B' = (rm2 B)
          in B := !B'; lam_ptr end
        | rm2 (var_ptr as ref (VarG(x,n,(_,_,visited as (ref false),_)))) =
          ( visited := true;
            var_ptr )
        | rm2 visited = (* we've been here before *) visited
in (let val _ = mk_graph_visited false ptr
    val ptr_out = rm2 ptr
    in ptr_out end
) end
```

```
(* These are support routines for implementing Wadsworth backpointers. I use
   "unique" integer IDs, from -100 downwards, called "binder IDs." This avoids
   interference with binding indices, which are positive or (for constants) small
   negative numbers. A hack, to be sure.
   next_ID generates "fresh" binderID's, needed when copying graphs. The functions
   bidx_to_bndrIDs_T : int → int → Term → Term  and
   bndrIDs_to_bidx : int → int → Term → Term  convert between binding indices and
   binderID's in graphs and Terms. The function
   chg_bndrIDs : int → int → Gnode ref → unit  replaces one binderID with another in a
   graph.
   All three of these functions produce intermediate structures used by term2graph
   (page 67) and graph2termT (page 67). They have no other use.
   I do not use ML refs, because, in practice, they make conversion back and forth to
   binding indices unpleasant.
*)
and next_ID () = (* generate ID's sequentially *)
( next_binder_ID := (!next_binder_ID - 1);
  !next_binder_ID )

and bndrIDs_to_bidx lev bndrID (App(M,N)) =
    App(bndrIDs_to_bidx lev bndrID M, bndrIDs_to_bidx lev bndrID N)
  | bndrIDs_to_bidx lev bndrID (Sus(B,P,n)) =
    Sus(bndrIDs_to_bidx (lev+1) bndrID B, bndrIDs_to_bidx (lev+1) bndrID P, n)
  | bndrIDs_to_bidx lev bndrID (Lam(B,n)) =
    Lam(bndrIDs_to_bidx (lev + 1) bndrID B, n)
  | bndrIDs_to_bidx lev bndrID (Var(bi,vmk,n)) =
    Var(if bndrID = bi then lev else bi,vmk,n)

and bidx_to_bndrIDs_T lev bndrID (App(M,N)) =
    App(bidx_to_bndrIDs_T lev bndrID M, bidx_to_bndrIDs_T lev bndrID N)
  | bidx_to_bndrIDs_T lev bndrID (Sus(B,P,n)) =
    Sus(bidx_to_bndrIDs_T (lev+1) bndrID B,bidx_to_bndrIDs_T (lev+1) bndrID P,n)
  | bidx_to_bndrIDs_T lev bndrID (Lam(B,n)) =
    Lam(bidx_to_bndrIDs_T (lev + 1) bndrID B, n)
  | bidx_to_bndrIDs_T lev bndrID (Var(bi,vmk,n)) =
    Var(if bi = lev then bndrID else bi,vmk,n)

and chg_bndrIDs o_bndrID n_bndrID (ref (AppG(M,N,indir,_))) =
    if !indir then chg_bndrIDs o_bndrID n_bndrID M
    else (chg_bndrIDs o_bndrID n_bndrID M;chg_bndrIDs o_bndrID n_bndrID N)
  | chg_bndrIDs o_bndrID n_bndrID (ref (LamG(B,_,_,_))) =
    chg_bndrIDs o_bndrID n_bndrID B
  | chg_bndrIDs o_bndrID n_bndrID (ref (VarG(si,n,_))) =
    if o_bndrID = (!si) then si := n_bndrID
    else ()
```

Bibliography

[1] Samson Abramsky and R. Sykes. SECD-m: A virtual machine for applicative multiprogramming. In FPCA '85 [110], pages 81–98. Cited on page 48.

[2] Luigia Carlucci Aiello and Gianfranco Prini. An efficient interpreter for the lambda-calculus. *Journal of Computer and System Sciences*, 23(3):383–424, December, 1981.

[3] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1989. Cited on page 109.

[4] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings Publishing Co., Redwood City, CA, 1989. Cited on page 103.

[5] Makoto Amamiya. Data flow computing and parallel reduction machine. *Future Generations Computing Systems*, 4(1):53–67, August, 1988. Cited on page 50.

[6] Paul Anderson, Chris L. Hankin, Paul Kelly, Peter E. Osmon, and Malcolm J. Shute. COBWEB-2: Structured specification of a wafer-scale supercomputer. In PARLE '87 [57], pages 51–67. Cited on page 50.

[7] Paul Anderson, Paul Kelly, and Phil Winterbottom. The feasibility of a general-purpose parallel computer using WSI. In PARLE '89 [157], pages 251–258. Cited on page 50.

[8] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In FPCA '87 [111], pages 301–324. Cited on page x.

[9] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of computation in functional language implementations. In High-Level Architecture '84 [92], pages 5.1–5.12. Cited on pages 25, 27, 38, 76, 98, 100, and 160.

[10] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel processing. In Fasel and Keller [65], pages 336–369. Cited on page 118.

[11] Lennart Augustsson and Thomas Johnsson. The <ν,G>-machine. In *Functional Programming Languages and Computer Architecture (FPCA)*, London, England, September 11–13, 1989. ACM Press (with Addison-Wesley). Cited on page 50.

[12] John Backus. The history of FORTRAN I, II, and III. In Richard L. Wexelblat, editor, *History of Programming Languages. Proceedings of the ACM SIGPLAN Conference*, pages 25–74. Academic Press, June 1–3, 1978.

[13] John W. Backus. Programming language semantics and closed applicative languages. In *1st ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–86, Boston, MA, October 1–3, 1973. Cited on pages 117 and 151.

[14] John W. Backus. Can programming be liberated from the von-Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August, 1978. 1977 Turing Award lecture. Cited on pages 6, 13, and 103.

[15] John W. Backus. The algebra of functional programs: Function level reasoning, linear equations, and extended definitions. In J. Díaz and I. Ramos, editors, *Formalization of Programming Concepts. Proceedings of an International Colloquium*, volume 107 of *Lecture Notes in Computer Science*, pages 1–43, Peniscola, Spain, April 19–25, 1981. Springer-Verlag. Cited on page 5.

[16] John W. Backus. From function level semantics to program transformation. In Hartmut Ehrig, C. Floyd, Maurice Nivat, and J. Thatcher, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 60–91, Berlin, Germany, March 25–29, 1985. Springer-Verlag. Cited on page 4.

[17] John W. Backus, John H. Williams, and Edward L. Wimmers. FL language manual (preliminary version). Research Report RJ 5339 (54809), IBM Almaden Research Center, San Jose, CA, November 7, 1986. Cited on page 5.

[18] H. P. Barendregt. *Lambda Calculus. Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, The Netherlands, revised edition, 1984. Cited on pages 12, 13, 19, and 22.

[19] Henk P. Barendregt, J. Richard Kennaway, Jan Willem Klop, and M. Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75(3):191–231, December, 1987. Cited on page 28.

[20] Henk P. Barendregt, M. C. J. D. van Eekelen, John R. W. Glauert, J. Richard Kennaway, M. J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In PARLE '87 [58], pages 141–158. Cited on page 51.

[21] Henk P. Barendregt, M. C. J. D. van Eekelen, John R. W. Glauert, J. Richard Kennaway, M. J. Plasmeijer, and M. Ronan Sleep. Towards an intermediate language based on graph rewriting. In PARLE '87 [58], pages 158–175. Cited on pages 51 and 180.

[22] Henk P. Barendregt, M. C. J. D. van Eekelen, M. J. Plasmeijer, John R. W. Glauert, J. Richard Kennaway, and M. Ronan Sleep. LEAN: An intermediate language based on graph rewriting. *Parallel Computing*, 9(2):163–177, January, 1989. Revised version of [21]. Cited on page 51.

[23] Henk P. Barendregt, M. C. J. D. van Eekelen, M. J. Plasmeijer, Pieter H. Hartel, L. O. Hertzberger, and Willem G. Vree. The Dutch Parallel Reduction Machine project. *Future Generations Computing Systems*, 3(4):261–270, December, 1987. Cited on pages 7, 49, and 49.

[24] C. Gordon Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw-Hill, New York, NY, 1971. Cited on page 182.

[25] Klaus Berkling. Head order reduction: A graph reduction scheme for the operational lambda calculus. In Fasel and Keller [65], pages 27–48. Cited on pages 19, 20, 45, 46, and 156.

[26] K[laus] J. Berkling. Reduction languages for reduction machines. In *2nd International Symposium on Computer Architecture (ISCA)*, pages 133–140, Houston, TX, January 20–22, 1975. IEEE Computer Society Press. Cited on page 117.

[27] Klaus J. Berkling. A symmetric complement to the lambda calculus. Technical Report ISF-76-7, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Bonn, W. Germany, September 14, 1976. Cited on pages 20 and 42.

[28] Klaus J. Berkling. Computer architecture for correct programming. In *5th International Symposium on Computer Architecture (ISCA)*, pages 78–84, Palo Alto, CA, April 3–5, 1978. IEEE Computer Society Press. Cited on page 117.

[29] Klaus J. Berkling. Experiences with integrating parts of the GMD-Reduction-Languages machine. In Brian Randell and Philip C. Treleaven, editors, *VLSI Architecture. An Advanced Course*, pages 381–394, Bristol, UK, July 19–30, 1982. University of Bristol, Prentice-Hall International. Cited on page 117.

[30] Klaus J. Berkling. The pragmatics of combinators. Technical Report 8803, CASE Center, Syracuse University, Syracuse, NY, February, 1988. Cited on page 46.

[31] Klaus J. Berkling and Elfriede Fehr. A consistent extension of the lambda calculus as a base for functional programming languages. *Information and Control*, 55(1–3):89–101, October–December, 1982. Cited on page 42.

[32] Klaus J. Berkling and Elfriede Fehr. A modification of the λ-calculus as a base for functional programming. In M. Nielsen and E. M. Schmidt, editors, *9th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 140 of *Lecture Notes in Computer Science*, pages 35–47, Aarhus, Denmark, July 12–16, 1982. Springer-Verlag. Cited on pages 20 and 42.

[33] D. I. Bevan, G. L. Burn, R. J. Karia, and J. D. Robson. Principles for the design of a distributed memory architecture for parallel graph reduction. *Computer Journal*, 32(5):461–469, October, 1989. Cited on page 49.

[34] David I. Bevan, Geoffrey L. Burn, and Rajeev J. Karia. Overview of a parallel reduction machine project. In PARLE '87 [57], pages 394–413. Cited on page 50.

[35] Guy E. Blelloch and James J. Little. Parallel solutions to geometric problems on the scan model of computation. In David H. Bailey, editor, *Proceedings of the 1988 International Conference on Parallel Processing (ICPP)—Vol. III Algorithms and Applications*, pages 218–222, University Park, PA, August 15–19, 1988. Pennsylvania State University Press. Cited on page 110.

[36] H. J. Boom. Lazy variable-renumbering makes substitution cheap. *Information Processing Letters*, 29(5):229–232, November 24, 1988. Cited on page 22.

[37] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Publishing Co., Reading, MA, 1975.

[38] T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer. Clean, a language for functional graph rewriting. In FPCA '87 [111], pages 364–384. Cited on page 51.

[39] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Report to the U. S. Ordnance Department, 1946. Reprinted in Bell and Newell, 1971 [24]. Cited on page 1.

[40] Geoffrey L. Burn. Developing a distributed memory architecture for parallel graph reduction. In *CONPAR88: Conference on Algorithms and Hardware for Parallel Processing*, UMIST, Manchester, UK, September 12–16, 1988. Cambridge University Press. Cited on page 49.

[41] G[eoffrey] L. Burn. Overview of a parallel reduction machine project II. In PARLE '89 [157], pages 385–396. Cited on page 49.

[42] F. Warren Burton and Matthew M. Huntbach. Virtual tree architectures. *IEEE Transactions on Computers*, C-33(3):278–280, March, 1984. Cited on page 118.

[43] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In FPCA '81 [68], pages 187–194. Cited on page 118.

[44] Michel Castan, Guy Durrieu, Bernard Lecussan, Michel Lemaître, Alessandro Contessa, Eric Cousin, and Paulino Ng. Toward the design of a parallel graph reduction machine: The MaRS project. In Fasel and Keller [65], pages 161–180. Cited on pages 50 and 50.

[45] Yaohan Chu, Leonard Haynes, Lee W. Hoevel, Arthur Speckhard, Edward A. Stohr, and Ralph H. Sprague, Jr., editors. *19th Hawaii International Conference on System Sciences (HICSS)*, volume I, Honolulu, HI, January 7–10, 1986. Cited on pages 194 and 197.

[46] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, NJ, 1941. Cited on pages 5 and 12.

[47] T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean, and A. C. Norman. SKIM—the S,K,I reduction machine. In *Conference Record of the 1980 LISP Conference*, pages 128–135, Stanford University, Palo Alto, CA, August 25–27, 1980. Cited on page 50.

[48] A. Contessa, E. Cousin, C. Couset, M. Cubero-Castan, G. Durrieu, B. Lecussan, M. Lemaître, and P. Ng. MaRS, a combinator graph reduction multiprocessor. In PARLE '89 [157], pages 176–192. Cited on page 50.

[49] Martin D. Cripps, Anthony J. Field, and Michael J. Reeve. An introduction to ALICE: A multiprocessor graph reduction machine. In Susan Eisenbach, editor, *Functional Programming: Languages, Tools, and Architectures*, Series in Computers and their Applications, pages 111–127. Ellis Horwood Ltd., 1987. Cited on pages 7 and 45.

[50] Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69(1–3):188–254, April–June, 1986. Cited on page 98.

[51] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. John Wiley and Sons, 1986. Cited on page 99.

[52] Haskell B. Curry and R. Feys. *Combinatory Logic*, volume I. North Holland, Amsterdam, 1958. Cited on page 22.

[53] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume II. North Holland, Amsterdam, 1972. Cited on page 22.

[54] William J. Dally and D. Scott Wills. Universal mechanisms for concurrency. In PARLE '89 [157], pages 19–33. Cited on page 7.

[55] Scott Danforth. DOT, a distributed operating system model of a tree-structured multiprocessor. In Howard J. Siegel and Leah Siegel, editors, *International Conference on Parallel Processing (ICPP)*, pages 194–201, Columbus, OH, August 23–26, 1983. Department of Computer and Information Sciences, Ohio State University. Cited on page 105.

[56] John Darlington and Michael J. Reeve. ALICE—a multi-processor reduction machine for the parallel evaluation of applicative languages. In FPCA '81 [68]. Cited on pages 7 and 45.

[57] J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors. *PARLE: Parallel Architectures and Languages Europe. Volume I: Parallel Architectures. Proceedings*, volume 258 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 15–19, 1987. Springer-Verlag. Cited on pages 178, 181, and 193.

[58] J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors. *PARLE: Parallel Architectures and Languages Europe. Volume II: Parallel Languages. Proceedings*, volume 259 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 15–19, 1987. Springer-Verlag. Cited on pages 180 and 180.

[59] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34(5):381–392, 1972. Cited on page 20.

[60] Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 65(2,3):122–157, May–June, 1985. Cited on page 51.

[61] Susan Dickey, Richard Kenner, Marc Snir, and Jon Solworth. A VLSI combining network for the NYU Ultracomputer. In *Proceedings of the IEEE International Conference on Computer Design (ICCD '85)*, pages 110–113, Port Chester, NY, October 7–10, 1985. Cited on page 110.

[62] R. Kent Dybvig. *Three Implementation Models for Scheme.* Ph.D. dissertation, University of North Carolina at Chapel Hill, 1987. Technical Report 87-011. Cited on pages 104 and 151.

[63] J. P. Eckert. Univac-Larc, the next step in computer design. In *Proceedings of the Easter Joint Computer Conference. Theme: New Devel-*

opments in Computers, pages 16–20, New York, NY, December 10–12, 1956. Cited on page 2.

[64] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In FPCA '87 [111], pages 34–45. Cited on page 100.

[65] Joseph H. Fasel and Robert M. Keller, editors. *Graph Reduction. Proceedings of a Workshop*, volume 279 of *Lecture Notes in Computer Science*, Santa Fe, NM, September 29–October 1, 1986. LANL, MCC, Springer-Verlag. Cited on pages 179, 180, 182, 186, 190, and 199.

[66] Anthony J. Field and Peter G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, Reading, MA, 1988. Cited on pages 5, 98, and 99.

[67] Michael J. Flynn. Some computer organizations and their effectiveness. *Computer Journal*, C-21(9):948–960, September, 1972. Cited on page 3.

[68] *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture (FPCA)*, Portsmouth, New Hampshire, October 18–22, 1981. Cited on pages 182, 184, and 193.

[69] Geoffrey A. Frank. *Virtual Memory Systems for Closed Applicative Language Interpreters.* Ph.D. dissertation, University of North Carolina at Chapel Hill, 1979. Cited on pages 108, 116, and 117.

[70] Geoffrey A. Frank, William E. Siddall, and Donald F. Stanat. Virtual memory schemes for an FFP machine. In High-Level Architecture '84 [92], pages 8.37–8.45. Cited on pages 108, 116, 116, and 117.

[71] R. E. Genner, J. L. Gustafson, and R. E. Montry. Development and analysis of scientific applications programs on a 1024-processor hypercube. Technical Report 88-0317, Sandia National Laboratories, Albuquerque, NM, February, 1988. Cited on page 3.

[72] S. Gill. Parallel programming. *Computer Journal*, 1(1):2–10, April, 1958. Cited on page 2.

[73] John R. W. Glauert, Kevin Hammond, J. Richard Kennaway, M. Ronan Sleep, G. W. Somner, Nicholas P. Holt, Michael J. Reeve, and Ian Watson. Extensions to Core Dactl 1. Technical report, Declarative

Systems Project, School of Information Systems, University of East Anglia, 1987. Cited on page 51.

[74] John R. W. Glauert, Nicholas P. Holt, J. Richard Kennaway, M. J. Reeve, and M. Ronan Sleep. An active term rewrite model for parallel computation. Document, Alvey DACTL Group, February, 1985. Cited on page 51.

[75] John R. W. Glauert, Nicholas P. Holt, J. Richard Kennaway, Michael J. Reeve, M. Ronan Sleep, and Ian Watson. DACTL0: A computational model and an associated compiler target language. Technical report, University of East Anglia, May 7, 1985. Cited on page 51.

[76] John R. W. Glauert, J. Richard Kennaway, and M. Ronan Sleep. Categorical descriptions of graph rewriting and garbage collection (in preparation), 1987. Cited on page 51.

[77] John R. W. Glauert, J. Richard Kennaway, and M. Ronan Sleep. DACTL: A computational model and compiler target language based on graph reduction. *ICL Technical Journal*, 5(3):509–537, May, 1987. Also Internal Report SYS-C87-03, Declarative Systems Project, University of East Anglia, December 10, 1987. Cited on page 51.

[78] John R. W. Glauert, J. Richard Kennaway, M. Ronan Sleep, Nicholas P. Holt, Michael J. Reeve, and Ian Watson. Specification of core Dactl 1. Internal Report SYS-C87-09, Declarative Systems Project, School of Information Systems, University of East Anglia, March 26, 1987. Cited on page 51.

[79] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In Fasel and Keller [65], pages 53–93. Cited on page 119.

[80] Joseph A. Goguen and José Meseguer. Software for the Rewrite Rule Machine. In ICOT '88 [107], pages 628–637. Cited on page 119.

[81] Benjamin F. Goldberg. Detecting sharing of partial applications in functional programs. In FPCA '87 [111], pages 408–425. Cited on page 98.

[82] Allan Gottlieb. An overview of the NYU Ultracomputer project. In J. J. Dongarra, editor, *Experimental Parallel Computing Architectures*,

volume 1 of *Special Topics in Supercomputing*, pages 25–95. North-Holland, 1987. Cited on page 110.

[83] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May, 1988. Cited on page 3.

[84] Dorothy Lobrano Guth, editor. *Letters of E. B. White.* Harper and Row, New York, NY, 1976. Cited on page x.

[85] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In Lisp/FP '84 [138], pages 9–17. Cited on page 48.

[86] Robert H. Halstead, Jr. An assessment of MultiLisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, December, 1986. Cited on page 48.

[87] Chris L. Hankin, Peter E. Osmon, and Malcolm J. Shute. COBWEB—a combinator reduction architecture. In FPCA '85 [110], pages 99–112. Cited on page 50.

[88] Peter G. Harrison and Hessam Khoshnevisan. Efficient compilation of linear recursive functions into object level loops. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 207–218, Palo Alto, CA, June 25–27, 1986. ACM, ACM SIGPLAN. Cited on pages 5, 7, and 45.

[89] Pieter H. Hartel. A comparative study of three garbage collection algorithms. PRM Project Internal Report D-23, Department of Computer Systems, University of Amsterdam, Amsterdam, The Netherlands, February, 1988. Cited on page 156.

[90] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *3rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 95–103, Atlanta, GA, January 19–21, 1976. Cited on page 39.

[91] L. O. Hertzberger and W[illem] G. Vree. A coarse grain parallel architecture for functional languages. In PARLE '89 [157], pages 269–285. Cited on pages 7 and 49.

[92] *Proceedings of the International Workshop on High-Level Computer Architecture 84*, Los Angeles, CA, May 21–25, 1984. Department of Computer Science, University of Maryland, College Park, MD. Cited on pages 178, 185, and 193.

[93] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985. Cited on pages 3, 110, and 118.

[94] W. Daniel Hillis. The Connection Machine. *Scientific American*, 256(6):108–115, June, 1987. Cited on page 118.

[95] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December, 1986. Cited on page 118.

[96] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986. Cited on pages 13, 78, 82, 85, and 90.

[97] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985. Cited on page 3.

[98] S. Holmstrom. A simple and efficient way to handle large data structures in applicative languages. In *Proceedings of the Declarative Programming Workshop*, pages 185–188, University College, London, April 11-13, 1983. Cited on page 118.

[99] F. Hommes. The heap/substitution concept—an implementation of functional operations on data structures for a reduction machine. In ISCA '82 [108], pages 248–256. Cited on page 117.

[100] F. Hommes, Werner E. Kluge, and Heinz Schlütter. A reduction machine architecture and expression oriented editing. Technical Report ISF 80.04, Gesellschaft für Mathematik und Datenverarbeitung (GMD), 1980. Cited on page 117.

[101] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. In *12th ACM Symposium on Principles of Programming Languages (POPL)*, pages 300–314, New Orleans, LA, January 14–16, 1985. Cited on page 118.

[102] Paul Hudak and Benjamin F. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transactions on Computers*, C-34(10):881–891, October, 1985. Cited on page 48.

[103] Paul Hudak and Benjamin F. Goldberg. Serial combinators: "optimal" grains of parallelism. In FPCA '85 [110], pages 362–399. Cited on page 48.

[104] Paul Hudak and Eric Mohr. Graphinators and the duality of SIMD and MIMD. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (Lisp/FP)*, pages 224–234, Snowbird, UT, July 25–27, 1988. Cited on pages 119 and 119.

[105] R. John M. Hughes. Super-combinators: A new implementation method for applicative languages. In Lisp/FP '82 [137], pages 1–10. Cited on pages 22, 26, 40, and 98.

[106] [R.] J[ohn] [M.] Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April, 1989. Cited on page 4.

[107] *ICOT '88. Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December, 1988. Cited on pages 186 and 192.

[108] *9th International Symposium on Computer Architecture (ISCA)*, Austin, Texas, April 26–29, 1982. Cited on pages 188 and 199.

[109] *12th International Symposium on Computer Architecture (ISCA)*, Boston, MA, June 17–19, 1985. Cited on pages 199 and 200.

[110] Jean-Pierre Jouannaud, editor. *Functional Programming Languages and Computer Architecture (FPCA)*, volume 201 of *Lecture Notes in Computer Science*, Nancy, France, September 16–19, 1985. Springer-Verlag. Cited on pages 178, 187, 189, 195, and 196.

[111] Gilles Kahn, editor. *Functional Programming Languages and Computer Architecture (FPCA)*, volume 274 of *Lecture Notes in Computer Science*, Portland, Oregon, September 14–16, 1987. Springer-Verlag. Cited on pages 178, 182, 185, 186, 196, and 200.

[112] Alan H. Karp. Programming for parallelism. *Computer*, 20(5):43–57, May, 1987. Cited on page 2.

[113] Takuya Katayama. Treatment of big values in an applicative language HFP. In Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, editors, *RIMS Symposia on Software Science*

and Engineering. Proceedings, volume 147 of *Lecture Notes in Computer Science*, pages 35–48, Kyoto, Japan, October, 1982. Springer-Verlag. Cited on page 118.

[114] Robert M. Keller, Frank C. H. Lin, and Jiro Tanaka. Rediflow multiprocessing. In *Intellectual Leverage: The Driving Technologies. Digest of Papers. 28th COMPCON, Spring 84*, pages 410–417, San Francisco, CA, February 27–March 1, 1984. Cited on pages 7, 45, 46, and 48.

[115] Robert M. Keller, Gary Lindstrom, and Suhas Patil. A loosely-coupled applicative multi-processing system. In *AFIPS Conference Proceedings, 1979 National Computer Conference*, pages 613–622, June, 1979. Cited on pages 47, 48, and 48.

[116] Robert M. Keller, Jon W. Slater, and Kevin T. Likes. Overview of Rediflow II development. In Fasel and Keller [65], pages 203–214. Cited on pages 7, 48, and 48.

[117] J. N. Kellman. Parallel execution of functional programs. Technical Report UCLA-ENG-83-02, UCLA Computer Science Department, Los Angeles, CA, 1983. Cited on page 106.

[118] J. Richard Kennaway. An outline of some results of Staples on optimal reduction orders in replacement systems. Technical Report CSA/19/1984, School of Information Systems, University of East Anglia, Norwich, England, March 20, 1984. Cited on page 94.

[119] J. Richard Kennaway. The correctness of an implementation of "functional" Dactl by non-atomic rewriting. Draft, Declarative Systems Project, School of Information Systems, University of East Anglia, June, 1987. Cited on page 51.

[120] J. Richard Kennaway. Implementing term rewrite languages in Dactl. In M. Dauchet and Maurice Nivat, editors, *CAAP '88. 13th Colloquium on Trees in Algebra and Programming*, volume 299 of *Lecture Notes in Computer Science*, pages 102–116, Nancy, France, March 21–24, 1988. Springer-Verlag. Cited on page 51.

[121] J. R[ichard] Kennaway and M. R[onan] Sleep. The 'language first' approach. In Fred B. Chambers, David A. Duce, and Gillian P. Jones, editors, *Distributed Computing*, number 20 in APIC Studies in Data

Processing, pages 111–124. Academic Press, London, UK, 1984. Cited on pages 97 and 99.

[122] Richard Kennaway and Ronan Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, October, 1988. Reviewed CR 8903-0130. Cited on page 98.

[123] Jan Willem Klop. Term rewriting systems: A tutorial. Note CS-N8701, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, May, 1987. Also in *Bulletin of the EATCS*, No. 32, 1987. Cited on page 51.

[124] Werner E. Kluge. The architecture of a reduction machine hardware model. Technical Report ISF 79.03, Gesellschaft für Mathematik und Datenverarbeitung (GMD), August, 1979. Cited on page 117.

[125] Werner E. Kluge. Cooperating reduction machines. *IEEE Transactions on Computers*, C-32(11):1002–1012, November, 1983. Cited on pages 117 and 117.

[126] Werner E. Kluge and Heinz Schlütter. An architecture for direct execution of reduction languages. In *Proceedings of the International Workshop on High-Level Computer Architecture*, Ft. Lauderdale, FL, May 27–28, 1980. Cited on page 117.

[127] Philip J. Koopman, Jr. and Peter Lee. A fresh look at combinator graph reduction (or, having a TIGRE by the tail). In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 110–119, Portland, OR, June 21–23, 1989. Cited on pages 50, 98, and 100.

[128] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10):965–968, October, 1985. Cited on page 110.

[129] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, October, 1988. Cited on page 110.

[130] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January, 1964. Cited on pages 6, 14, and 98.

[131] P. J. Landin. A correspondence between ALGOL 60 and Church's lambda-notation (2 parts). *Communications of the ACM*, 8(2,3):89–101,158–165, February–March, 1965. Cited on page 12.

[132] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March, 1966. Cited on page 12.

[133] Sany Leinwand and Joseph Goguen. Architectural options for the Rewrite Rule Machine. In Lana P. Kartashev and Steven I. Kartashev, editors, *Supercomputing '87. Proceedings of the Second International Conference on Supercomputing*, volume III, pages 63–70, 1987. Cited on page 119.

[134] Sany Leinwand and Joseph A. Goguen. A Rewrite Rule Machine: Architectural options and testbed facilities for the Rewrite Rule Machine (final report). SRI Project ECU 1243, SRI, July, 1986. Cited on page 119.

[135] Sany Leinwand, Joseph A. Goguen, and Timothy C. Winkler. Cell and ensemble architecture for the Rewrite Rule Machine. In ICOT '88 [107], pages 869–878. Cited on page 119.

[136] Frank C. H. Lin and Robert M. Keller. Gradient model: A demand-driven load balancing scheme. In *6th International Conference on Distributed Computing Systems (ICDCS)*, pages 329–336, Cambridge, MA, May 19–23, 1986. IEEE Computer Society Press. Cited on page 48.

[137] *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming (Lisp/FP)*, Pittsburgh, PA, August 15–18, 1982. Cited on pages 189 and 193.

[138] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming (Lisp/FP)*, Austin, Texas, August 6–8, 1984. Cited on pages 187 and 198.

[139] *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (Lisp/FP)*, Cambridge, MA, August 4–6, 1986. Cited on pages 196, 197, and 198.

[140] Gyula Magó and David Middleton. The FFP Machine—a progress report. In High-Level Architecture '84 [92], pages 5.13–5.25. Cited on pages 103 and 110.

[141] Gyula A. Magó. A network of microprocessors to execute reduction languages (2 parts). *International Journal of Computer and Information Sciences*, 8(5 and 6):349–385 and 435–471, 1979. Cited on pages 103 and 104.

[142] Gyula A. Magó. Copying operands versus copying results: A solution to the problem of large operands in FFP's. In FPCA '81 [68], pages 93–97. Cited on page 115.

[143] Gyula A. Magó. Data sharing in an FFP Machine. In Lisp/FP '82 [137], pages 201–207. Cited on page 116.

[144] Gyula A. Magó. Making parallel computation simple: The FFP Machine. In *Technological Leverage: A Competitive Necessity. Digest of Papers. 30th COMPCON, Spring 85*, pages 424–428, San Francisco, CA, February 25–28, 1985. Cited on pages 103 and 117.

[145] Gyula A. Magó. Bibliography of UNC faculty and students on functional programming and the FFP machine, April 4, 1988. Cited on page 104.

[146] Gyula A. Magó and Will[iam D.] Partain. Implementing dynamic arrays: A challenge for high-performance machines. In Lana P. Kartashev and Steven I. Kartashev, editors, *Supercomputing '87. Proceedings of the Second International Conference on Supercomputing*, volume I, pages 491–493, 1987. Cited on pages 105 and 154.

[147] Gyula A. Magó and Donald F. Stanat. The FFP Machine. In Veljko M. Milutinović, editor, *High-Level Language Computer Architectures*, pages 430–468. Computer Science Press, 1989. Cited on pages 103, 104, 106, 109, and 125.

[148] D. L. McBurney and M. Ronan Sleep. Transputer-based experiments with the ZAPP architecture. In PARLE '87 [57], pages 242–259. Cited on page 118.

[149] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April, 1960. Cited on pages 12 and 98.

[150] James R. McGraw and Timothy S. Axelrod. Exploiting multiprocessors: Issues and options. In Robert G. Babb II, editor, *Programming Parallel Processors*, pages 7–25. Addison-Wesley Publishing Co., Reading, MA, 1988. Cited on page 2.

[151] Henk Meijer and Selim G. Akl. Optimal computation of prefix sums on a binary tree of processors. *International Journal of Parallel Programming*, 16(2):127–136, April, 1988. Cited on page 109.

[152] David Middleton. Alternative program representations for the FFP Machine. In K. Waldschmidt and B. Myhrhaug, editors, *Microcomputers, Usage and Design. 11th EUROMICRO Symposium on Microprocessing and Microprogramming.*, pages 85–93, Brussels, Belgium, September 3–6, 1985. Cited on pages 105 and 141.

[153] David Middleton and Bruce T. Smith. FFP machine support for language extensions. In HICSS '86 [45], pages 59–66. Cited on pages 104 and 107.

[154] G. Mitschke. *Eine algebraische Behandlung von λ-K-Kalkül und Kombinatorischer Logik.* Ph.D. dissertation, Rheinischen Friedrich-Wilhelms Universität, Bonn, W. Germany, 1970. Cited on page 98.

[155] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for lambda Prolog. Submitted for publication, March, 1989. Cited on page 90.

[156] Masayuki Numao and Masamichi Shimura. Evaluation of graph representations with active nodes. In Eiichi Goto, Keijiro Araki, and Taiichi Yuasa, editors, *RIMS Symposia on Software Science and Engineering II. Proceedings of the Symposia 1983 and 1984*, volume 220 of *Lecture Notes in Computer Science*, pages 17–43, Kyoto, Japan, 1984. Springer-Verlag. Cited on page 50.

[157] Eddy Odijk, Martin Rem, and Jean-Claude Syre, editors. *PARLE '89: Parallel Architectures and Languages Europe. Volume I: Parallel Architectures. Proceedings*, volume 365 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, June 12–16, 1989. Springer-Verlag. Cited on pages 178, 182, 183, 183, 187, and 196.

[158] John O'Donnell. Supporting functional and logic programming languages through a data parallel VLSI architecture. In José G. Delgado-Frias and Will R. Moore, editors, *VLSI for Artificial Intelligence (Proceedings of an International Workshop)*, pages 49–60, Oxford, England, July, 1988. Kluwer Academic Publishers. Cited on page 118.

[159] John T. O'Donnell. An architecture that efficiently updates associative aggregates in applicative programming languages. In FPCA '85 [110], pages 164–189. Cited on page 118.

[160] John T. O'Donnell, Timothy Bridges, and Sidney W. Kitchel. A VLSI implementation of an architecture for applicative programming. *Future Generations Computing Systems*, 4(3):245–254, October, 1988. Cited on page 118.

[161] Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, MA, 1985. Cited on page 96.

[162] Michael J. O'Donnell and Robert I. Strandh. Toward a fully parallel implementation of the lambda calculus. Technical Report JHU/EECS-84/13, Johns Hopkins University, 1984. Cited on page 96.

[163] Rex L. Page and Linda S. Barasch. Parallel computation, functional programming, and Fortran 8x. In Michael T. Heath, editor, *Hypercube Multiprocessors, 1986: Proceedings of the 2nd Conference on Hypercube Multiprocessors*, pages 57–69, Knoxville, TN, August 26–27, 1985. SIAM.

[164] G. W. Petznick. *Combinatory Programming.* Ph.D. dissertation, University of Wisconsin, Madison, WI, 1970. Cited on page 22.

[165] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987. Cited on pages 6, 16, 18, 18, 19, 19, 38, 42, 49, 148, 153, 158, 162, and 164.

[166] Simon L. Peyton Jones. FLIC—a functional language intermediate code. *ACM SIGPLAN Notices*, 23(8):30–48, August, 1988. Cited on page 51.

[167] S[imon] L. Peyton Jones. Parallel implementations of functional programming languages. *Computer Journal*, 32(2):175–186, April, 1989. Cited on pages 49 and 49.

[168] Simon L. Peyton Jones, Chris Clack, and Jon Salkild. High-performance parallel graph reduction. In PARLE '89 [157], pages 193–206. Cited on page 48.

[169] Simon L. Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP—a high performance architecture for parallel graph reduction. In FPCA '87 [111], pages 98–112. Cited on page 48.

[170] David A. Plaisted. An architecture for fast data movement in the FFP Machine. In FPCA '85 [110], pages 147–163. Cited on pages 106 and 151.

[171] David A. Plaisted. An architecture for functional programming and term rewriting. In J. Vivian Woods, editor, *Fifth Generation Computer Architectures. Proceedings of the IFIP TC 10 Working Conference on Fifth Generation Computer Architectures*, pages 221–234, Manchester, U.K., July 15–18, 1985. North-Holland. Cited on pages 8, 119, 151, and 151.

[172] Jerry L. Potter, editor. *The Massively Parallel Computer.* MIT Press, Cambridge, MA, 1985. Cited on page 3.

[173] John D. Ramsdell. The CURRY chip. In Lisp/FP '86 [139], pages 122–131. Cited on page 50.

[174] György Révész. Axioms for the theory of lambda-conversion. *SIAM Journal on Computing*, 14(2):373–382, May, 1985. Cited on pages 21, 94, 96, and 97.

[175] György E. Révész. An extension of lambda-calculus for functional programming. *Journal of Logic Programming*, 1(3):241–251, October, 1984. Cited on page 97.

[176] G[yörgy] E. Révész. *Lambda-Calculus, Combinators, and Functional Programming*, volume 4 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988. Cited on pages 94, 96, 97, 97, and 156.

[177] György E. Révész. Parallel graph-reduction with a shared memory multiprocessor system. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988. Cited on page 97.

[178] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, January, 1973. Cited on page 98.

[179] Mark Scheevel. NORMA: A graph reduction processor. In Lisp/FP '86 [139], pages 212–219. Cited on page 50.

[180] Heinz Schlütter. *Investigations into the Foundations of Functional Programming and an Implementation of Existential Quantification on a Lambda Calculus Based Reduction Machine.* Ph.D. dissertation, CASE Center, Syracuse University, Syracuse, NY, May, 1987. Available in revised form as CASE Center Technical Report No. 8714 (August 1987). Cited on page 42.

[181] Claudia Schmittgen. A data type architecture for reduction machines. In HICSS '86 [45], pages 78–87. Cited on page 117.

[182] Dana S. Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata, 21*, pages 19–46, Polytechnic Institute of Brooklyn, 1971. Also report PRG-6, Oxford University Computing Laboratory. Cited on page 12.

[183] Raj K. Singh and Vernon L. Chi. The design of a memory management subsystem for the FFP Machine. Technical Report TR 89-017, University of North Carolina at Chapel Hill, May 1, 1989. Version 1.0. Cited on page 105.

[184] M. Ronan Sleep and J. Richard Kennaway. The zero assignment parallel processor (ZAPP) project. In David A. Duce, editor, *Distributed Computing Systems Programme*, volume 5 of *IEE Digital Electronics and Computing Series*, pages 250–269. Peter Peregrinus Ltd., 1984. Cited on page 118.

[185] Bruce T. Smith. Logic programming on an FFP Machine. In *1984 International Symposium on Logic Programming (SLP)*, pages 177–186, Atlantic City, NJ, February 6–9, 1984. IEEE Computer Society Press. Cited on page 104.

[186] Bruce T. Smith. *Logic Programming on an FFP Machine.* Ph.D. dissertation, University of North Carolina at Chapel Hill, 1989. In preparation. Cited on pages 104 and 108.

[187] Bruce T. Smith and David Middleton. Exploiting fine-grained parallelism in production systems. In R. Goebel, editor, *Proceedings of the 7th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, pages 262–270, Edmonton, Alberta, June 6–10, 1988. Cited on pages 104 and 107.

[188] John Staples. A class of replacement systems with simple optimality theory. *Bulletin of the Australian Mathematical Society*, 17(3):335–350, 1977. Cited on page 94.

[189] John Staples. Optimal reduction in replacement systems. *Bulletin of the Australian Mathematical Society*, 16(3):341–349, 1977. Cited on page 94.

[190] John Staples. A graph-like lambda calculus for which leftmost-outermost evaluation is optimal. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph-Grammars and their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 440–454, Bad Honnef, W. Germany, October 30–November 3, 1978. Springer-Verlag. Cited on pages 94, 94, 94, 96, and 98.

[191] John Staples. A lambda calculus with naive substitution. *Journal of the Australian Mathematical Society (Series A)*, 28(Part 3):269–282, November, 1979. Cited on page 22.

[192] John Staples. Computation on graph-like expressions. *Theoretical Computer Science*, 10(2):171–185, February, 1980. Cited on page 94.

[193] John Staples. Optimal evaluations of graph-like expressions. *Theoretical Computer Science*, 10(3):297–316, March, 1980. Cited on page 94.

[194] John Staples. Speeding up subtree replacement systems. *Theoretical Computer Science*, 11(1):39–47, May, 1980. Cited on page 94.

[195] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine LISP: Fine-grained parallel symbolic processing. In Lisp/FP '86 [139], pages 279–297. Cited on page 118.

[196] W. R. Stoye, T. J. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In Lisp/FP '84 [138], pages 197–204. Cited on page 50.

[197] Kenneth R. Traub. An abstract parallel graph reduction machine. In ISCA '85 [109], pages 333–341. Cited on page 50.

[198] Philip C. Treleaven and Richard P. Hopkins. A recursive computer for VLSI. In ISCA '82 [108], pages 229–238. Cited on page 118.

[199] David A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, pages 31–49, January, 1979. Cited on pages 22, 98, and 119.

[200] W[illem] G. Vree. Experiments with coarse-grain parallel graph reduction. *Future Generations Computing Systems*, 4(4):299–306, March, 1989. Cited on pages 49 and 49.

[201] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* D. Phil. thesis, University of Oxford, September 1, 1971. Cited on pages 6, 9, 12, 19, 25, 31, 31, 42, 42, 54, and 92.

[202] Ian Watson, John Sargeant, Paul Watson, and J. Vivian Woods. The cost of parallel graph reduction (draft). FLAGSHIP Project internal report, 1987. Cited on page 46.

[203] Ian Watson, John Sargeant, Paul Watson, and Viv Woods. Flagship computational models and machine architecture. Technical report, Dept. of Computer Science, University of Manchester, 1986. Cited on page 47.

[204] Ian Watson and Paul Watson. Graph reduction in a parallel virtual memory environment. In Fasel and Keller [65], pages 265–274. Cited on pages 47, 47, and 50.

[205] Ian Watson, Viv Woods, Paul Watson, Richard Banach, Mark Greenberg, and John Sargeant. Flagship: A parallel architecture for declarative programming. In *15th International Symposium on Computer Architecture (ISCA)*, pages 124–130, Honolulu, HI, May 30–June 3, 1988. Cited on pages 45, 46, and 46.

[206] Paul Watson. *Parallel Reduction of Lambda Calculus Expressions.* Ph.D. dissertation, University of Manchester, Manchester, England, 1986. Cited on pages 46, 99, and 156.

[207] Paul Watson and Nicholas P. Holt. Extended graph reduction as a general purpose parallel computation model (issue no. 1). Flagship Project Internal Report FS/MU/PW/024-88, University of Manchester, Manchester, England, November 1, 1988. Submitted to PARLE 89. Cited on page 47.

[208] Paul Watson and Ian Watson. Evaluating functional programs on the FLAGSHIP machine. In FPCA '87 [111], pages 80–97. Cited on pages 46 and 46.

[209] Peter Wegner. *Programming Languages, Information Structures, and Machine Organization.* McGraw-Hill, 1968. Cited on page 6.

[210] Colin Whitby-Strevens. The transputer. In ISCA '85 [109], pages 292–300. Cited on pages 3 and 49.

[211] A. Wikströom. *Functional Programming Using Standard ML.* Prentice-Hall International, Englewood Cliffs, NJ, 1988. Cited on page 166.

[212] Wayne T. Wilner. Recursive machines. Internal report, Xerox PARC, 1980. Cited on page 118.

[213] John M. Wozencraft and Arthur Evans, Jr. Notes on programming linguistics. Class notes for MIT EE course 6.231, Programming Linguistics, July, 1969. Cited on page 28.

Index

Entries in a sans serif font refer to functions in the ML programs; a page number in italics says where the function is defined. Authors of cited works may be traced through the bibliography.

AAL, *see* absolute application level
ABL, *see* absolute binding level
absolute application level, *see also* level numbers
absolute binding level, *see also* level numbers
absolute index, *see also* level numbers
absolute nesting level, *see also* level numbers
abstractions, 13
active L-segments, 107
AIX, *see* absolute index
α-conversion, 16
ANL, *see* absolute nesting level
application level, *see also* level numbers
applications, 14
applicative-order evaluation, 19
archetypal problem, 9
associative matching, 125

backpointers, 34
β-conversion, 16
β-expansion, 16
β-normal form, 17
β-reduction, 15
β_s rule, 63
β_s-reduction, 63
bidx_to_bndrIDs_T, 67, *177*
binder-IDs, 34
binders, 13
    scope, 13
binding index, 20
binding level, *see also* level numbers
binding path, 14
bndrIDs_to_bidx, 67, *177*
BNF, *see* β-normal form
body
    λ-abstraction, 13
    suspension, 58
bound variables, 13

chg_bndrIDs, 41, *177*
chk_vars, 61, 72, 79, 168, 169, *170*, 172
closure-based reduction, 100
closures, 98
colored parentheses, *see* FFP Machine (partitioning)
combinators, 22
computation sharing, 27
computational models, 7

constants, 14
conversion, 16
copying
  necessary, 27
  speculative, 27
  unnecessary, 27
cumulative-sum operations, *see* FFP Machine (scan operations)

data parallelism, 3
de Bruijn numbers, *see* variables (binding index)
director strings, 99
Dix, Dorothea Lynde, iv
dynamic arrays, 105

eager evaluation, 19
environments, 98
Erskine, John, of Carnock, 53
η-conversion, 16
η-expansion, 16
η-reduction, 16
eval_BNF, 19, 22, *25*, 70, 74
eval_LF, 22, 25, 69, 74
Eval_WBNF, 22
eval_WBNF, *25*, 54
evaluation orders, 17
  innermost spine reduction, 164
  normal order, 5, 17
  safe, 18, 19
  unsafe, 19
expansion, 16
exponential blow-up, 28
extensionality, 16

FFP Machine
  communication
    broadcasting, 108
    message wave, 108
    sorting, 108
  dynamic arrays, 105
  global networks, 106
  L-segments, 106
    active, 107
  reduction routines, 105
  scan operations, 109
  selectors, 113
  storage management, 105
FFPM, *see* FFP Machine
fixed-program machines, 6
Follow*fill pointers, 126
free variables, 14
fully lazy copying, 38
functional programming, 4

g-connections, 34
GAM, *see* global addressable memory
Gnode, *33*, 166
graph reduction
  parallel, 7
graph rewriting, 51
graph2termT, 35, *67*, 177

head-normal form, 18
higher-order functions, 4
HNF, *see* head-normal form

implicit parallelism, 6
incr_bd_vars, 88, *171*
incr_free_vars1, 23, 25, 72, *171*, 172
incr_free_vars2, 62, 79, 88, *171*
incr_refcnt, 36, 67, *175*, 176
incr_var_range, *171*
index, *see also* level numbers
indirection nodes, 34
is_app, *169*
is_bd_follow_fill_ptr, 72, *172*
is_bd_follow_ptr, 72, *172*
is_bd_var_or_ptr, 61, 79, *172*

is_higher_up_follow_ptr, 72, *172*
is_lam, *169*
is_ptr, 61, *169*
is_well_formed, 57, *61*, 61

Jeeves, Reginald, 163

L-array, 104
L-segments, 106
λ-abstractions, 13
λ-applications, 14
λ-calculus, 5, 12
lambda form, 19
λ-terms
  abstractions, 13
  applications, 14
  binders, 13
  bound variables, 13
  combinators, 22
  constants, 14
  free variables, 14
  supercombinators, 22
  variables, 13
λ-trees, 14
λ-application lists, *155*
λ-calculus rules
  α-conversion, 16
  β-conversion, 16
  β-expansion, 16
  β-reduction, 15
  η-conversion, 16
  η-expansion, 16
  η-reduction, 16
[λ-up] rule, 82
λ_g-equivalence, 34
λ_g-interpreter, 31
λ_s-interpreter, *53*
λ_s-interpreter, 53
λ_s-pointers, 58
λ_s-terms, 58
λ_{sg}-equivalence, 63
λ_t-equivalence, 21
last-instance relocation, 75
lazy copying, 38
lazy evaluation, 4, 18
lazy_copy, 32, 35, 36, 40, *41*, 76
level numbers, 110
  absolute, 111
  application, 112
  binding, 112
  indices, 110, 111
  nesting, 111
  relative, 111
LF, *see* lambda form
linear expansion, 61

maximal free expressions, 38
message wave, 108
MFE, *see* maximal free expressions
mk_graph_visited, 37, *175*, 176
mod_vars, 72, 168–170, *171*, 172, 173

name capture, 20
name-free λ-calculus, 14
necessary copying, 27
nesting level, *see also* level numbers
next_ID, 67, *177*
normal forms
  β-normal form, 17
  head-normal form, 18
  root-lambda form, 19, 74
  weak β-normal form, 19
  weak head-normal form, 19
normal-order evaluation, 5, 17

onestepG, 32, 35, *36*, 54, 64, 66
onestepS, 54, 57, 63, 71, *72*, 74, 161
onestepT, 22, *23*, 54, 66

parallel-prefix operations, *see* FFP Machine (scan operations)
parse tree, 6
plain nodes, 34, 57
plain_equivs, 21, *23*
pointee, 58
pointers, 27, 58
ptrize_bd_vars, 72, *173*

RAL, *see* relative application level
rand, 14
rator, 14
RBL, *see* relative binding level
recurring example, 22, 40, 90, 138
reduction, 5
reduction routines, 105
reduction rules, 5
reentrant machines, 6
relative application level, *see also* level numbers
relative binding level, *see also* level numbers
relative index, *see also* level numbers
relative nesting level, *see also* level numbers
RIX, *see* relative index
RLF, *see* root-lambda form
rm_indir_nodes, 32, 78, *176*
RNL, *see* relative nesting level
root path, 14
root-lambda form, 19, 74

s-connection, downward, 86
s-connection, upward, 85
s-connections, 62
safe evaluation orders, 19
scans, 109, *see* FFP Machine (scan operations)
scope of a binder, 13
selectors, 113
set_subbed, 36, 67, *175*
SFE, *see* substituted free expressions
sg-connections, 62
sharing, 9
  computation, 27
  space, 27
  trivial, 27
sharing analysis, 98
space sharing, 27
speculative copying, 27
spine of a λ-tree, 18
std_subst, 23, 25, 62, 88, *172*
storage management, *see* FFP Machine (storage management)
subst, 72, 169, *172*
substG, 32, 35, 36, *37*, 67
substituted free expressions, 39
subterms, 13
supercombinators, 22
suspension lists, *154*
suspension pointee, 58
suspensions, 58
swap_levs, 88, *173*

task hints, 126
Term, *59*, 166
term rewriting, 51
term2graph, 66, *67*, 177
term2termT, 61, *62*
tidying, 86
  local, *130*
  nonlocal, *130*
tidying rules, 78
tidyterm, 57, 87, *88*, 173
toplevG, 32, *32*, 35
toplevS, *57*, 57
trashpickup, 57, *79*, 79, 87

tree reduction, 106
trivial sharing, 27
trivial suspensions, 80

unnecessary copying, 27
unsafe evaluation orders, 19

variables, 13
  binding index, 20
  binding path, 14
  name capture, 20
  renaming, 16, 20
  root path, 14

WBNF, *see* weak β-normal form
weak beta-normal form, 19
weak head-normal form, 19
WHNF, *see* weak head-normal form

xapping, 118