Optimal Interpreters for Lambda-calculus Based Functional Languages

by

Vinod Kumar Kathail

B.Tech, Bhopal University, Bhopal, India
(1976)
M.Tech, Indian Institute of Technology, Kanpur, India
(1978)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1990

© Massachusetts Institute of Technology 1990

Signature of Author______________________________
Department of Electrical Engineering and Computer Science
25 May 1990

Certified by______________________________
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

ARCHIVES
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
AUG 10 1990
LIBRARIES

Optimal Interpreters for Lambda-calculus Based Functional Languages

by

Vinod Kumar Kathail

Submitted to the Department of Electrical Engineering and Computer Science
on 25 May 1990
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

The λ-calculus is the basis of a number of practical programming languages, especially lazy functional languages such as LML, Miranda and Haskell. Efficient interpreters for the λ-calculus can serve as a basis for designing efficient interpreters and partial-evaluators for these languages. Interpreters for the λ-calculus are also useful in other areas, *e.g.*, formal specification systems, automated theorem proving.

In this thesis, we develop an *optimal* interpreter for the λ-calculus. The interpreter is a practical realization of J.-J. Lévy's theoretical specification of an optimal reduction strategy for the λ-calculus. The basic idea underlying the interpreter is a new graph representation for λ-expressions that permits sharing of not only subexpressions but also *contexts*, *i.e.*, parts of an expression that are not complete subexpressions. This is in contrast to the commonly used representations of expressions, which permit sharing of only subexpressions.

We describe the graph representation, the set of reduction rules used to transform graphs and the reduction strategy used to apply the rules. The set of rules includes a version of the β-rule as well as certain other rules, some of which are similar to the rules for handling environments in an environment-based interpreter for the λ-calculus. All the reduction rules are local, constant-time operations on graphs. We give translations between λ-expressions in De Bruijn's notation and our graph representation. The input to the interpreter as well as output of the interpreter are "clean" representations of λ-terms. The reduction strategy for applying the reduction rules is quite simple. We have implemented a version of the interpreter on lisp machines.

To prove the correctness of the interpreter, we develop two calculii, called λ_{fc} calculus and λ_f calculus. λ_{fc} calculus is essentially the term version of the graph reduction system underlying the interpreter. λ_f calculus is obtained from λ_{fc} calculus by removing certain types of terms and reduction rules that are not very useful for terms. We show the correspondence between the graph reduction system underlying the interpreter and λ_{fc} calculus as well as correspondence between the two calculii and De Bruijn notation. Although λ_f calculus was motivated by the interpreter, it may be of general interest because of the way it simulates changing of De Bruijn numbers. We prove that the interpreter implements Lévy's specification of an optimal reduction strategy for the λ-calculus.

We also strengthen a result of Barendregt *et al* that states that if λ-expressions are represented as trees, then there is no *recursive* (one-step) reduction strategy that is optimal. Our result

provides some justification for the basic assumption underlying the optimality criterion, *i.e.*, the number of β-contractions performed in reducing in expression is a good measure of the cost of reducing the expression.

Thesis Supervisor: Arvind

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

It is a pleasure to acknowledge my thesis supervisor, Professor Arvind, for his advice, support and encouragement during my long tenure at MIT. He helped me transform my ideas into a form that can be generally understood. He also helped me finish on schedule, for which I am grateful.

I would like to thank my thesis readers, Professors Albert Meyer and Rishiyur S. Nikhil, for reading various drafts of my thesis, usually on short notice. I am especially thankful to Nikhil for his careful reading of a draft of my thesis and his suggestions to improve it.

I would like to thank Pierre-Louis Curien and Keshav Pingali for their contribution to this thesis. They helped me simplify both the algorithm as well as its presentation. I especially appreciate Curien's support and encouragement.

I would like to thank Jean-Jacques Lévy and John Staples with whom I discussed my work on several occasions.

My thanks to Prof. Thomas Cheatham who provided support and gave me something else to do to take my mind off the thesis.

My thanks to Zena Ariola for helping me prepare slides and thesis drafts when help was critically needed, to Andy Boughton for his help in preparing the thesis, and to members of CSG (past and present) for general moral support.

I am grateful to my parents for their love and support. They have waited for a long time for me to graduate. I am also grateful to my brothers, sister, and their families. I only wish my brother, Arvind, was here to see me graduate.

To my daughter, Ragini, who made the life more pleasant, my love.

Finally, my thanks and love to my wife, Renu. Without her support and patience, this thesis would not have been possible.

In Memory of Arvind Kumar Kathail

Contents

1 Introduction **10**
1.1 The λ-calculus and its interpreters: Basics . . . 11
1.1.1 Expressions and reduction rules . . . 11
1.1.2 Sequential interpreters for the λ-calculus . . . 14
1.2 Optimality Criterion . . . 16
1.2.1 Minimum length reduction beginning with an expression . . . 17
1.2.2 Non-existence of interpreters that find minimum length reductions . . . 18
1.2.3 Sharing of subexpressions . . . 22
1.2.4 What is an optimal interpreter? . . . 24
1.2.5 Designing an optimal interpreter . . . 26
1.3 Basic techniques for implementing λ-calculus . . . 27
1.4 Organization of the thesis . . . 30

2 Lévy's theory and its implications **31**
2.1 Overview of Lévy's theory . . . 31
2.2 Lévy's labelled λ-calculus . . . 34
2.3 Optimal reductions . . . 36
2.4 Difficulties in implementing Lévy's theory . . . 37
2.4.1 Properties of redexes to be shared . . . 37
2.4.2 Sharing and copying of applications . . . 38
2.4.3 Avoiding capture of free variables . . . 40
2.5 Summary . . . 41

3 An Optimal Interpreter for the λ-calculus **42**
3.1 The problem with (ordinary) graph reduction . . . 42
3.2 The idea of sharing contexts . . . 44
3.3 A representation for sharing contexts . . . 46
3.4 Reducing a β-redex . . . 51
3.5 Translation procedure . . . 55
3.6 Graphs may be cyclic . . . 56
3.7 Converting subgraphs representing β-redexes into β-redexes: New reduction rules 57
3.8 Computation rule . . . 61
3.9 Deficiencies and improvements . . . 62
3.10 New structure of control environments . . . 64
3.11 Summary . . . 69

4 Optimal Interpreter using De Bruijn's Notation **70**
4.1 De Bruijn's Notation for the λ-calculus . . . 70
4.2 λ-splitting and β-reduction in De Bruijn's notation . . . 74
4.3 A notation for graph reduction . . . 78
4.4 The interpreter . . . 85
4.4.1 Functions on control environments . . . 86
4.4.2 λ_{fc} graphs . . . 87
4.4.3 Translations . . . 91
4.4.4 Reduction rules . . . 93
4.4.5 Reduction Strategy . . . 99
4.4.6 Summary . . . 101

5 λ_f Calculus **104**
5.1 Functions on control environments . . . 104
5.2 λ_f terms and E reduction rules . . . 110
5.3 Translations between λ_f calculus and De Bruijn notation . . . 112
5.4 Traces of subexpressions relative to $E \cup F$ reduction . . . 114
5.5 Free and bound occurrences of i . . . 114
5.6 β-reduction . . . 116
5.7 β_f reduction commutes with $E \cup F$ reductions . . . 117
5.8 Equivalence of λ_f calculus and De Bruijn notation . . . 126
5.9 Reduction strategies . . . 131
5.10 Summary . . . 132

6 Proofs of Correctness and Optimality **133**
6.1 Introduction . . . 133
6.2 λ_{fc} calculus . . . 137
6.2.1 Functions on control environments . . . 137
6.2.2 λ_{fc} terms and E_c reduction rules . . . 138
6.2.3 Translation of λ_{fc} terms into λ_f terms . . . 141
6.2.4 Traces for $E_c \cup F_c$ reduction . . . 143
6.2.5 Free and bound occurrences . . . 143
6.2.6 β reduction . . . 144
6.2.7 Traces for β_{fc} and λ-sp . . . 144
6.2.8 Correspondence between λ_{fc} calculus and De Bruijn Notation . . . 144
6.3 Term approximations of term graphs . . . 150
6.4 Proof of Correctness . . . 154
6.4.1 Correspondence between the graph reduction system underlying the interpreter and λ_{fc} calculus . . . 157
6.4.2 Correctness of the interpreter . . . 171
6.5 Proof of Optimality . . . 172
6.6 I . . . 184

7 Conclusions **185**
7.1 Comparison with Related work . . . 188
7.2 Some ideas for further research . . . 188

A Extension of the result of Barendregt *et. al.* **190**

List of Figures

1.1 Reduction of M using Wadsworth's graph reduction technique 23
1.2 Reduction of the leftmost redex in M 23

2.1 Reduction graph of M in Example 2.1 32
2.2 Schematic representation of N to illustrate copying of λ-nodes 41

3.1 (a) Graph after the reduction of the leftmost redex in M (b) Graph after copying the shared abstraction and reducing the leftmost redex 43
3.2 (a) Graph after copying of Nodes 30 and 60 (b) Graph after the reduction of the leftmost redex 45
3.3 The graph after copying Node 50 46
3.4 (a) Expressions M and N (b) The context $C[]$ with hole filled with a conditional node (c) The graph representing M and N with only one copy of $C[]$ 47
3.5 (a) A context shared among more than two expressions (b) A shared context embedded inside another shared context 49
3.6 (a) Representation of M and N using functions on control environments (b) Illustration of the working of the translation procedure 50
3.7 (a) A β-redex whose abstraction part is shared (b) Graph after copying the root of the abstraction and occurrence of bound variable. Transforming the graph in (a) to the one in (b) is called λ-*splitting*. 52
3.8 Graph after reducing the redex in an obvious way 53
3.9 Correct way to reduce a β-redex 54
3.10 Graphs showing reduction of $\Omega \equiv (\lambda x.x\ x)\ (\lambda x.x\ x)$ 57
3.11 Simplified version of the graph in Figure 3.10 (b) 58
3.12 (a) A β-redex (b, c, d) Subgraphs that represent β-redexes but are not β-redexes 58
3.13 New reduction rules 60
3.14 Example to illustrate that copying performed in pushing an application node past a conditional node doesn't destroy optimality 62
3.15 Successive application of λ-splitting introduces several new control variables . . . 65
3.16 λ-splitting with new structure of control environments 66
3.17 β-reduction: combining control and environment variables 68

4.1 (a) Tree representation of M (b) Implicit bindings and reference depths (c) Term after naively reducing the leftmost redex in (b) 71
4.2 Relationship of the structure of control environments with the nesting structure of λ-abstraction 75
4.3 Notational changes 76
4.4 λ-splitting and β-reduction using DeBruijn's notation 76
4.5 λ-splitting and β-reduction using new encoding of n and binding pointers 79

4.6 The graph rewrite rule corresponding to the term rule for S combinator 82
4.7 (a) and (b) Situations in which homomorphisms between graphs exist. (c) A situation in which there in no homomorphism between graphs. 83
4.8 Applying a graph reduction rule 84
4.9 F rules 88
4.10 F rules (cont.) 89
4.11 Conditions satisfied by λ_{fc} graphs 90
4.12 Rules (λ-sp) and (β_{fc}) 94
4.13 Rules (ε-λ), (ε-c) and (Ap-c) 96
4.14 Rules to eliminate conditional nodes 97
4.15 Rules (ε-ε), (ε-ap) 98
4.16 Rules (β_g) and (ε-λ_g) 102
4.17 Rule (β'_g) 103

Chapter 1

Introduction

The main objective of this thesis is to develop an *optimal interpreter* for the λ-calculus and to give proofs of its correctness and optimality.

The λ-calculus, which was introduced by Church [9], can be viewed as a simple but powerful programming language based on the notion of function. It provides facilities for applying functions to arguments and for defining new functions. Functions are supported in full generality—they may be passed as arguments and returned as results. Its operational semantics mainly consists of a single rule (β-rule) that specifies how to evaluate a function application.

Because of its expressive power and the simplicity of its operational semantics, the λ-calculus has been used to study various aspects of programming languages. More importantly, a number of practical programming languages are explicitly based on the λ-calculus; examples include Lisp [25], ML [26], Lazy ML [2], Miranda [37], and Haskell [16]. In fact, most functional programming languages, especially the ones with *lazy* semantics such as Lazy ML and Miranda, are simply "sugared" form of the λ-calculus; they can be easily translated into λ-calculus augmented with a set of constants (see for example [27]). Backus's FP [3] is a notable exception; it is based on function composition and not on function application.

Our main motivation for designing an optimal interpreter for the λ-calculus is that such an interpreter can serve as the basis for designing optimal implementations for functional languages. Further, languages based on the λ-calculus are also used in other areas, *e.g.*, formal specification systems, automated theorem proving systems, and an optimal interpreter for the λ-calculus may be useful in these areas.

Intuitively, we are interested in the *most efficient* interpreter for the λ-calculus. But,

1. How do we measure the efficiency of an interpreter?

2. How do we actually show that a given interpreter is indeed the most efficient interpreter?

In this chapter, we answer these questions, define the notion of an optimal interpreter, and describe the main ideas underlying our interpreter. Most of the discussion in this chapter will be quite informal.

Section 1.1 is concerned with the basics. We describe expressions and reduction rules of the λ-calculus and discuss *sequential* interpreters for the λ-calculus. In Section 1.2, we introduce the notion of a minimum length reduction beginning with an expression; such a reduction can be viewed as the most efficient way of evaluating the expression. We show that for all practical purposes interpreters that realize minimum length reductions of expressions don't exist.

However, one is usually not interested in whether an interpreter finds a minimum length reduction beginning with an expression or not; one is more interested in the overall efficiency of an interpreter.

In the usual setting of the λ-calculus, expressions are represented as strings of symbols, or more properly as trees. Practical implementation of programming languages, however, rarely use the tree representations of expressions. Representations of expressions that permit *sharing of subexpressions,e.g.*, directed acyclic graph representations of expressions, are especially important in designing efficient interpreters for lazy functional languages. By using such a representation it is possible to design an interpreter, that in reducing an expression, matches the efficiency of a minimum length reduction beginning with the expression.

In Section 1.2.3, we show the advantages to be gained by using representations that permit sharing of subexpressions. In Section 1.2.4, we describe the notion of an optimal interpreter. Lévy gave a precise characterization of the kind of sharing of subexpressions that must be performed by an optimal interpreter. We will present Lévy's theory of optimal reductions in the next chapter. In Section 1.3, we review some of the existing techniques for implementing λ-calculus. None of these techniques, however, gives us an optimal interpreter for the λ-calculus. Section 1.4 gives the outline of the rest of the thesis.

1.1 The λ-calculus and its interpreters: Basics

1.1.1 Expressions and reduction rules

The λ-calculus consists of a set of expressions and a number of rules, called *reduction rules*, to simplify expressions. Expressions (or terms) of the calculus are built from a set of *variables*,

$x, y, z, \ldots$, using the following operations:

Application: An application is written simply as $(M\,N)$ where both M and N are expressions; M is the function part of the application and N is the argument part.

λ-abstraction: A λ-abstraction is written as $(\lambda x.M)$ where x is a variable and M is an expression, which may or may not contain occurrences of x. Intuitively, $(\lambda x.M)$ corresponds to a function with one formal parameter; x is the name of the formal parameter and M is the body of the function.

Parentheses may be omitted if the omission causes no ambiguity. Further, we follow the usual convention that application is left-associative; for example, the expression MNP stands for $((MN)P)$.

An important aspect of the λ-notation for writing functions is that the prefix λx in $\lambda x.M$ *binds* all those occurrences of x in M that are in its lexical scope.

Example 1.1 Consider the expression $(\lambda x.(\lambda x.\underline{x}\,y))\,\underline{x}$. The first occurrence of underlined x is bound by the inner λ-abstraction, whereas the second underlined occurrence of x is not bound by any λ-abstraction.

A variable occurrence in an expression is *bound* if it is bound by a λ-abstraction in the expression; otherwise it is *free*. Furthermore, names of bound variables are unimportant. We will consider expressions that differ in names of bound variables to be syntactically identical; for example, expressions $\lambda x.y\,x$ and $\lambda z.y\,z$ are identical. Sometimes this equivalence principle is explicitly stated as a rule, called α-rule, and expressions that differ only in the names of bound variables are called α-equivalent.

The central reduction rule is the β-rule. Intuitively, the rule states that to apply a function to an argument, replace all occurrence of the formal parameter of the function inside the body of the function by the actual parameter. To describe the β-rule, we will use the notation $M[x := N]$ for the result of *substituting* the expression N for free occurrences of variable x in the expression M.

$$(\beta): \qquad (\lambda x.M)\,N \longrightarrow M[x := N]$$

A subtle aspect of the substitution operation is that care must be taken to avoid "capture of free variables". That is, in the expression $M[x := N]$ free occurrences of variables in N should

not become bound by λ-abstractions inside M. The way to ensure that it doesn't happen is to rename λ-bound variables in M. The simplest way is to systematically rename all λ-bound variables inside M so that they are different from all free variables of N. The important point to note here is that a correct implementation of the substitution operation must avoid capture of free variables.

An expression of the form $(\lambda x.M)\, N$ is called a *β-redex*. The β-rule is applicable in any context. We write $M \longrightarrow N$ to denote that the expression M reduces to N by an application of the β-rule. The β-rule describes a single computation step that may be repeated to evaluate an expression. The evaluation process either will terminate in an expression that contains no β-redexes, called a *normal form*, or will continue indefinitely.

Example 1.2 Consider the expression

$$M \equiv D\,(I\,a) \quad \text{where } D \equiv \lambda x.x\,x \text{ and } I \equiv \lambda x.x.$$

We will use the λ-expressions I and D quite frequently. The expression M can be evaluated to normal form as follows (the redex reduced at each step is underlined):

$$\begin{aligned} M &\equiv \underline{D\,(I\,a)} \\ &\longrightarrow \underline{I\,a}\,(I\,a) \\ &\longrightarrow a\,\underline{(I\,a)} \\ &\longrightarrow a\,a \end{aligned}$$

In general, an expression may have several redexes. For this reason, there may be several ways to reduce an expression.

Example 1.2 (cont.) Another way to reduce M to normal form is to first reduce $(I\,a)$ to a and then apply D to a.

We write $M \twoheadrightarrow N$ if M can be reduced to N in zero or more reduction steps. A *reduction* is a sequence of the form $M_0 \xrightarrow{R_0} M_1 \xrightarrow{R_1} \cdots M_n$ where $R_0, R_1, \ldots$ identify redexes reduced at each step.

Two of the most important results about the λ-calculus are as follows. First, the λ-calculus has the *Church-Rosser* or *confluence property*. The Church-Rosser property is that if $M \twoheadrightarrow N$ and $M \twoheadrightarrow N'$, then there exists an expression P such that $N \twoheadrightarrow P$ and $N' \twoheadrightarrow P$. An important consequence of this property is that an expression either has no normal form or has an *unique* normal form. Thus, different ways of reducing an expression cannot produce different

normal forms. That is not to say that if an expression has a normal form, then all reductions starting from the expression reach the normal form; some may, some may not.

Example 1.3 Consider the expression

$$K\,a\,\Omega \quad \text{where } K \equiv \lambda x.\lambda y.x \text{ and } \Omega \equiv (\lambda x.x\,x)(\lambda x.x\,x).$$

The expression has the normal form a as shown by the following reduction:

$$\underline{K\,a}\,\Omega \longrightarrow \underline{(\lambda y.a)\,\Omega} \longrightarrow a$$

But, $\Omega \longrightarrow \Omega$. So the following reduction doesn't terminate.

$$K\,a\,\underline{\Omega} \longrightarrow K\,a\,\underline{\Omega} \longrightarrow \cdots$$

The second result gives a simple and uniform way to find normal forms. The result states that if an expression has a normal form, then the normal form can be found by repeatedly reducing the *leftmost* redex. Reductions in which the leftmost redex is reduced at each step are called *leftmost* or *normal order* reductions. For example, the first reduction in Example 1.3 is a leftmost reduction; it should come as no surprise that the reduction reaches the normal form of the expression.

Barendregt's book [4] is the standard reference for the λ-calculus; we mostly follow his notation. Introductory treatment of the λ-calculus can also be found in books by Stoy [35], Peyton-Jones [27].

1.1.2 Sequential interpreters for the λ-calculus

The job of an interpreter is to try to reduce a given λ-expression to normal form by repeatedly applying the β rule. Thus, an interpreter is simply an organized and perhaps efficient way of applying the β rule. We shall consider only *sequential* interpreters for the λ-calculus in this thesis.

The following recursive procedure describes the general structure of an interpreter. It takes a λ-expression to be evaluated as argument, and if it terminates, it returns the normal form of the input expression.

interpreter exp = **if** *exp* is in normal form **then** *exp*
else 1. Select a redex in *exp*
2. Reduce the selected redex to obtain a new expression,

say *exp'*

3. *interpreter exp'*

The first two operations in the else clause, namely selecting a redex and reducing the selected redex, can be viewed as an "elementary step", which is repeated until a normal form is reached. The selection of a redex at each step is usually governed by a rule, called *computation rule* [1]; most common examples are the leftmost rule and the leftmost-innermost rule.

In designing an interpreter, we are, of course, free to choose a computation rule. As we shall see later in the chapter, we are also free to represent expressions and implement the β rule any way we wish.

Correctness and efficiency will be our main concerns in designing an interpreter. The correctness of an interpreter can be divided into two parts: *partial correctness* and *termination*. An interpreter is partially correct if the result produced by the interpreter is indeed the normal form of the input expression. Partial correctness of an interpreter simply says that the interpreter implements the β rule correctly, for the Church-Rosser Theorem assures us that the order in which redexes are reduced is unimportant as far as normal forms are concerned. An interpreter that is partially correct and terminates on all expressions that have normal forms will be called a *normalizing* interpreter. This is a non-standard terminology; we prefer it, however, because many authors equate correctness with partial correctness [39]. In this thesis, we shall be interested mainly in normalizing interpreters for the λ-calculus.

Any discussion of efficiency requires that we have a way of measuring efficiency. The precise cost of reducing an expression using an interpreter can be obtained by summing up the cost of each of the elementary steps. Working with such a precise efficiency measure, however, is quite difficult, for the exact cost of an elementary step depends upon the size of the expression. Thus, we need a more abstract way of measuring the efficiency of an interpreter. We will have more to say about this later.

Choosing a computation rule

Choosing an appropriate computation rule is crucial in designing an interpreter, for both the correctness and the efficiency of an interpreter depends upon the computation rule it uses.

We are, of course, free to choose any computation rule. Most practical interpreters, however, use simple computation rules, *e.g.*, leftmost, leftmost-innermost. Mainly, we shall be interested

[1] A related notion is that of a *reduction strategy*, see cite Barendregt.

in computation rules that are "deterministic" and "easy to compute". Putting it another way, computation rules of interest are the ones that are total recursive functions on λ-expressions, and furthermore are relatively simple to compute. The reasons for these restrictions are mostly practical. The action of an interpreter on an expression should be repeatable. Further, each elementary step should take a small (or at least finite) amount of time.

Partial correctness of an interpreter doesn't depend upon the computation rule; the termination property, however, does. Consider the two most commonly used computation rules, namely leftmost and leftmost-innermost. The leftmost rule guarantees termination whereas the leftmost-innermost doesn't. For example, the expression in Example 1.3 has a normal form, but its leftmost-innermost reduction is infinite. Computation rules that guarantee termination will be called *normalizing* rules. Barendregt et al [6] discuss a family of normalizing computation rules, called spine reduction strategies.

Efficiency of an interpreter also depends upon the computation rule it uses. For example, the evaluation of an expression according to the leftmost rule usually takes more steps than according to the leftmost-innermost rule, provided the evaluation terminates in both cases. Consider Example 1.2 again. The reduction of M according to the leftmost rule takes 3 steps whereas its leftmost-innermost reduction only 2 steps.

Broadly speaking, normalizing rules lead to inefficient interpreters, although they are the only ones that guarantee termination. The reason is that normalizing rules usually select redexes whose reduction tend to create multiple copies of existing redexes. For example, the reduction of the leftmost redex in M of Example 1.2 duplicates the redex $I\,a$.

Thus, there seems to be a conflict between the two goals of correctness and efficiency in designing an interpreter.

1.2 Optimality Criterion

In this section, we introduce the notion of a minimum length reduction beginning with an expression; such a reduction can be viewed as the most efficient way of evaluating the expression. We show that for all practical purposes interpreters that realize minimum length reductions of expressions don't exist.

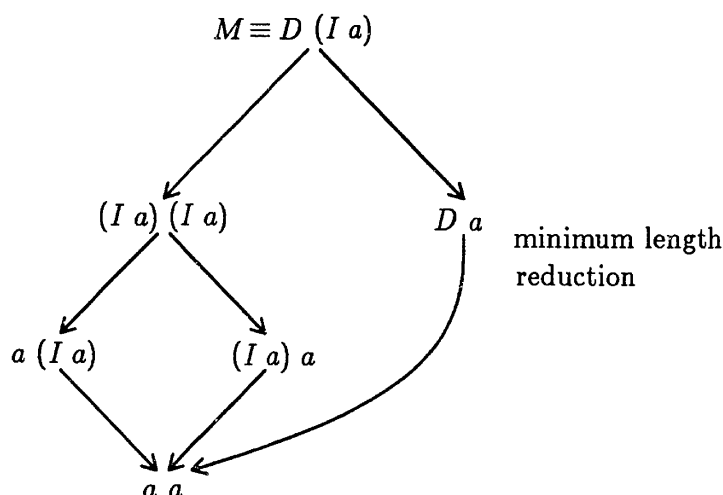
1.2.1 Minimum length reduction beginning with an expression

A λ-expression can be reduced in many ways. All reductions beginning with a term can be captured in a (possibly infinite) diagram, usually called the *reduction graph* of the expression. Essentially, the reduction graph of an expression describes all possible ways of evaluating an expression.

Example 1.4 Consider the expression M in Example 1.2, *i.e.*,

$$M \equiv D\,(I\,a) \quad \text{where } D \equiv \lambda x.x\,x \text{ and } I \equiv \lambda x.x.$$

The reduction graph of M is shown below.



Now, consider an expression that has a normal form. Then, we know that there is one reduction, namely leftmost reduction, that reaches the normal form. In general, there are other reductions that also reach normal form. Among all reductions to normal form, there is at least one reduction whose length, in number of β reductions, is no greater than the length of any other reduction. We call such a reduction a *minimum length reduction* to normal form.

In the above example, there are three reductions beginning with M, and all of them reach the normal form. The reduction on the left hand side of the diagram is the leftmost reduction starting with M whereas the reduction on the right hand side is the minimum length reduction to normal form. In this example, there is only one minimum length reduction to normal form. In general, however, there may be more than one such reduction.

If an expression has no normal form, then all reductions beginning with the expression are of infinite length. Abusing the terminology a bit, we can call any reduction to be a minimum length reduction beginning with the expression.

We assume that the cost of a reduction is proportional to its length. Then, a minimum length reduction beginning with an expression is the most efficient way to evaluate the expression.

1.2.2 Non-existence of interpreters that find minimum length reductions

In this section, we show that for all practical purposes interpreters that find minimum length reductions don't exist. We consider interpreters that operate in the usual framework of the λ-calculus, though we believe the result applies to any way of implementing the λ-calculus. In this setting, the only choice we have in designing an interpreter is that of the computation rule.

It is trivial to design an interpreter that finds minimum length reductions. The computation rule used by the interpreter operates as follows. At each step of the evaluation, the computation rule tries to construct a minimum length reduction beginning with the expression. If the expression has a normal form, then the construction will terminate. The computation rule then simply returns the redex reduced in the first step of the reduction. If the expression has no normal form, then the construction will not terminate. In this case the computation rule itself doesn't terminate. Such an interpreter, however, is of little practical interest. Indeed, interpreters using computation rules that may diverge can hardly be called interpreters. Usually, one considers only those interpreters in which each elementary step performs a small amount, or at least a finite amount, of work. Note that there is a distinction between an interpreter diverging because it performs an infinite number of steps and an interpreter diverging because an elementary step of the interpreter diverges.

There is no computation rule that finds minimum length reductions and always terminates. Essentially, any computation rule that finds minimum length reductions is of the form described above. The reason lies in the following two facts:

1. The reduction of a redex may *duplicate* other redexes because β reduction involves substituting a *copy* of the argument part of the redex for each occurrence of the bound variable. See Example 1.2; the reduction of the leftmost redex in M duplicates the redex $I\ a$.

2. The reduction of a redex may *erase* other redexes. See example 1.3; the reduction of the leftmost redex erases the redex Ω.

Duplication of a redex increases work, whereas reducing a redex that will be erased at a later step is wasted effort. If an expression has a normal form, then a reduction to normal form in which either of the two happens cannot be a minimum length reduction. Thus, a

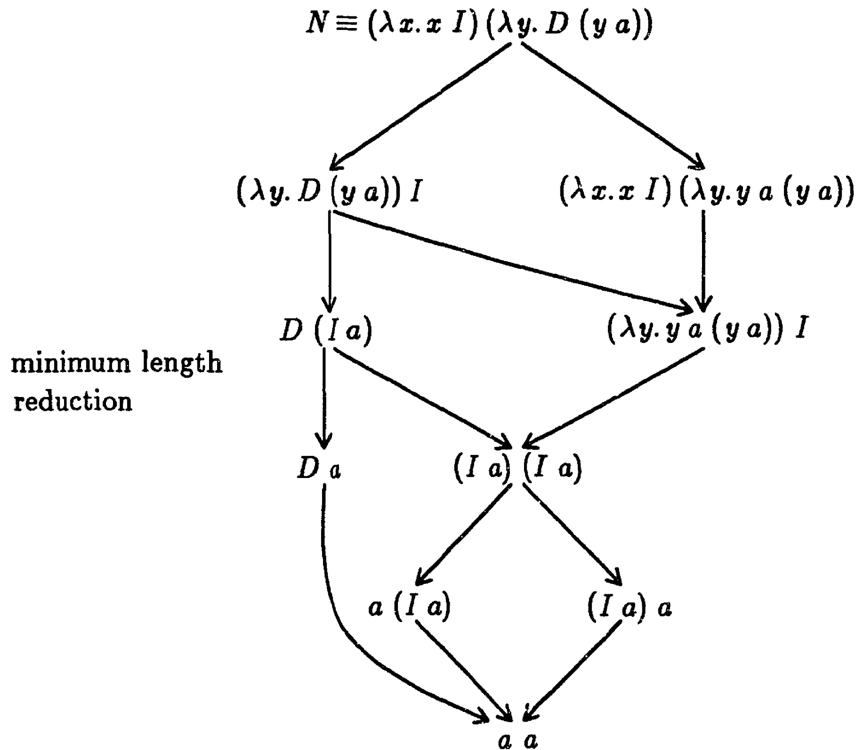
computation rule that finds minimum length reductions must select, at each step, a redex that does not duplicate other redexes and will not be erased at a later step. These are contradictory requirements. Redexes that will not be erased at a later step usually duplicate other redexes, and redexes that don't duplicate other redexes may be erased at a later step. Furthermore, there is no effective way to decide whether a redex will be erased at a later step or not.

Consider, for example, the two most common ways of selecting a redex, namely, outermost and innermost. An outermost redex, *e.g.*, the leftmost, can't be erased at a later step; that is why computation rules that select outermost redexes are normalizing. But it may duplicate other redexes. An innermost redex, on the other hand, doesn't duplicate other redexes. But it may get erased at a later step; that is why computation rules that select innermost redexes are non-normalizing. Moreover, even if an innermost redex doesn't get erased, it may not be the one that should be selected; the following example illustrates this fact.

Example 1.5 Consider the expression

$$N \equiv (\lambda x.x\, I)\,(\lambda y.D\,(y\, a)) \quad \text{where } D \equiv \lambda x.x\, x \text{ and } I \equiv \lambda x.x.$$

N contains exactly two redexes, the leftmost redex and the innermost redex $D\,(y\, a)$. Furthermore, the innermost redex doesn't get erased in any reduction to normal form. In the expression N, the leftmost redex and not the innermost redex should be selected because the minimum-length reduction from N to normal form begins with the leftmost redex; see the reduction graph of N given below.

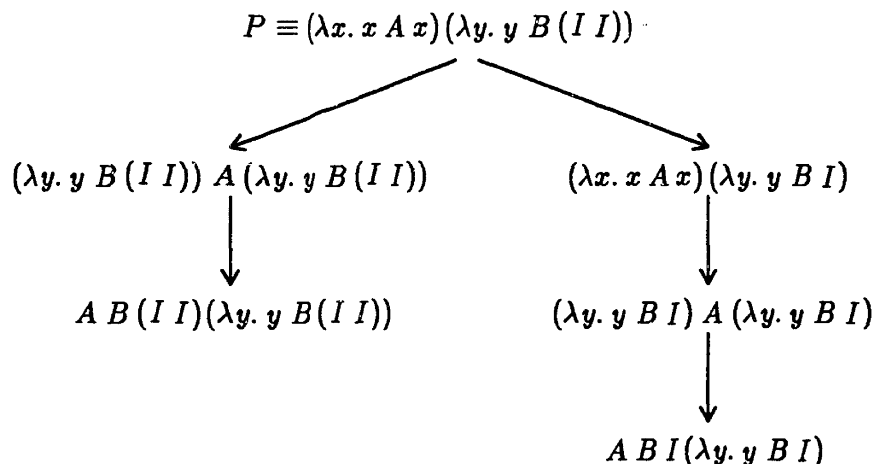


The following example shows even more clearly why there is no computation rule that finds minimum length reduction and always terminates. It is taken from [4, 5] where it is used in a formal proof of this fact.

Example 1.6 Consider the following expression

$$P \equiv (\lambda x.x\ A\ x)\ (\lambda y.y\ B\ (I\ I))$$

where A and B are λ-terms in normal form and $I \equiv \lambda x.x$. Like Example 1.5, The expression contains exactly two redexes, the leftmost redex and the innermost redex $I\ I$. The following diagram shows reductions relevant to our discussion.



The reduction on the left hand side begins with the leftmost redex whereas the reduction on the right hand side begins with the innermost redex. In both the reductions, we reach an expression that contains $A\,B$ as a subexpression. Now, consider the following two cases:

1. $A\,B \twoheadrightarrow \lambda x.\lambda y.I$. Then, we claim that a minimum-length reduction of P to normal form begins with the leftmost redex. Thus, a computation rule that finds minimum length reductions must select the leftmost redex.

2. $A\,B \twoheadrightarrow a$. Then, we claim that a minimum-length reduction of P to normal form begins with the innermost redex. Thus, a computation rule that finds minimum length reductions must select the innermost redex.

We leave it to the reader to verify the above claims. Thus, if a computation rule is optimal, it must be able to decide whether $A\,B$ reduces to $\lambda x.\lambda y.I$ or to a. But that, in general, is undecidable.

Summarizing the discussion, if we work in the usual framework of the λ-calculus, then there is no interpreter, other than the "degenerate" one, that finds minimum length reductions. We believe the result is true no matter how we implement the λ-calculus. The only way to find minimum length reductions is essentially the way outlined at the beginning of the section.

However, one is usually not interested in whether an interpreter finds a minimum length reduction beginning with an expression or not; one is more interested in the overall efficiency of an interpreter. Thus, an interpreter that, in reducing an expression, matches the efficiency of a minimum length reduction beginning with the expression is as good as an interpreter that finds a minimum length reduction. This change in viewpoint, however, doesn't change the situation much if we work in the usual setting of the λ-calculus. In this setting, the action of an interpreter determines a reduction beginning with the expression. Thus, an interpreter can match the efficiency of a minimum length reduction beginning with an expression only if the interpreter actually finds a minimum length reduction.

In the usual setting of the λ-calculus, expressions are represented as strings of symbols, or more properly as trees. By changing representation, however, it is possible to design an interpreter that, in reducing an expression, matches the efficiency of a minimum length reduction beginning with the expression. In the next section, we discuss the advantages to be gained by changing representation.

1.2.3 Sharing of subexpressions

The tree representation of expressions has the obvious disadvantage that the β reduction involves substituting a *copy* of the argument part for each occurrence of the bound variable. Copying of the argument part may result in the copying of existing redexes, and that is the main reason why normalizing interpreters that use the tree representation tend to be quite inefficient (see the end of Section 1.1). Copying of the argument part, however, seems unnecessary. It can be avoided by using a representation of expressions that permit sharing of subexpressions,*e.g.*, a directed acyclic graph representation. We will use the term ***graph representation*** to refer to a representation that permit sharing of subexpressions, since such a representation either explicitly or implicitly defines a graph structure on expressions.

Wadsworth [39] first proposed the idea of using a graph representation of expressions to design efficient interpreters for the λ-calculus. We go into his technique in some detail as it illustrates most clearly and easily the advantages to be gained from using a graph representation. We use our favorite expression M to illustrate the technique.

Example 1.7 Recall that

$$M \equiv D\,(I\,a) \quad \text{where } D \equiv \lambda x.x\,x \text{ and } I \equiv \lambda x.x.$$

Now consider the reduction of the leftmost redex in M. If we use the tree representation, then the argument part of the redex, *i.e.*, $I\,a$ is copied in two places. We can, however, keep only one copy of the the argument part and substitute ***pointers*** to it; see the first step of reduction in Figure 1.1. The graph in Figure 1.1(b) represents the expression

$$I\,a\,(I\,a),$$

which is obtained by simply unraveling the graph. This expression is, of course, the same as the one obtained by reducing the leftmost redex using the tree representation.

Now consider the graph in Figure 1.1(b). The subgraph rooted at Node 20 is a redex; it actually represents two redexes in the expression represented by the graph. To reduce this redex, we apply the β rule to the subgraph as usual. Further, we redirect all pointers to Node 20 to the result of reduction; that is we *update* the root of the redex with the result of the reduction. See the second reduction step in Figure 1.1.

The advantages offered by representing expressions as graphs should now be obvious. A subgraph which is a redex can represent a set of redexes in the expression represented by the

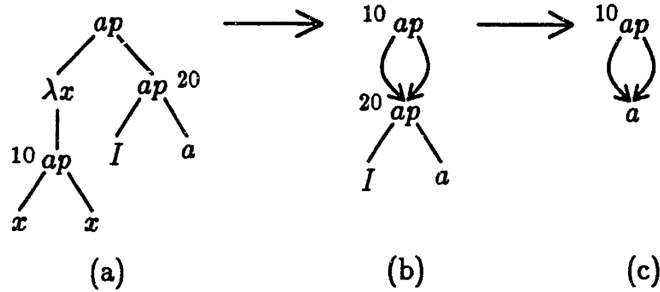


Figure 1.1: Reduction of M using Wadsworth's graph reduction technique

whole graph. Thus, by reducing one redex in a graph we can simultaneously reduce several redexes in the expression represented by the graph

The basic idea presented above needs further refinements. The main difficulty arises from the sharing of λ-abstractions.

Example 1.8 Consider the following expression.

$$Q \equiv ((\lambda x.x\, I(x\, J))\, (\lambda y.I\, (y\, a)), \text{ where } I \equiv \lambda x.x \text{ and } J \equiv \lambda x.a$$

The reduction of the leftmost redex in Q is obvious; the graph after the reduction is shown in Figure 1.2 In this graph, two redexes, namely the subgraphs rooted at Nodes 10 and 20 share

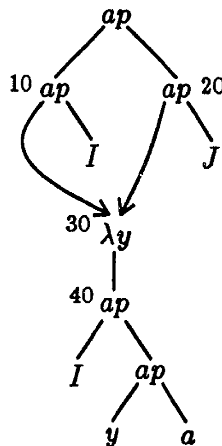


Figure 1.2: Reduction of the leftmost redex in M

a λ-abstraction. It is not clear how to reduce either of these redexes. Consider, for example, the reduction of the redex rooted at Node 10. The substitution of 3 for y directly would lead to an incorrect result. The solution is to create two copies of the shared abstraction. Copying

whole of the abstraction, however, is wasteful and unnecessary. Wadsworth proposed a rule to copy only parts of the λ-abstraction; see [39] (also [1]).

Graph representations of expressions are especially important in designing efficient normalizing interpreters. Normalizing computation rules usually select redexes whose argument part contain other redexes, and these are precisely the types of redexes for which the use of a graph representation is most beneficial.

Thus, we have the following strategy for designing efficient normalizing interpreters. First, we choose a suitable graph representation of expressions. Then, we take a simple normalizing computation rule (*e.g.*, leftmost) defined over the tree representation and adapt it to the graph representation. In this way, we retain the main advantage of normalizing rule, namely termination, and gain efficiency by avoiding the copying of redexes.

Is it possible to use the above strategy to design an interpreter that, in reducing an expression, matches the efficiency of a minimum length reduction beginning with the expression? It is not immediately obvious what the representation should be. Consider Example 1.7. The reduction shown in Figure 1.1, which is actually the leftmost reduction using a simple directed acyclic graph representation, has the same length as the minimum length reduction of M to normal form (see Example 1.4 for the reduction graph of M). On the other hand, copying of an existing redex seems unavoidable even if we use a graph representation of expressions. Consider the graph in Figure 1.2 (Example 1.8). As we noted, the redex rooted at Node 10 can be reduced only after copying all or parts of the shared λ-abstraction. Note, however, that the body of the λ-abstraction contains a redex, *i.e.*, the one rooted at Node 40. Any interpreter that copies this redex cannot match the efficiency of the minimum length reduction of Q to normal form. The best rule for copying parts of the shared λ-abstraction is the one given by Wadsworth, but his rule implies copying of this redex. As we shall see in the next section, the answer to the question posed at the beginning of the paragraph is indeed yes.

1.2.4 What is an optimal interpreter?

By now it should be apparent that what we are leading to is the following: An interpreter is optimal if the cost of reducing a λ-expression expression using the interpreter is no more than the cost of a minimum length reduction beginning with the expression. Of course, we have to specify how to find the cost in both cases.

Recall our assumption that the cost of a reduction is proportional to its length (see Section

1.2.1). Ignoring the constant of proportionality, the cost of a reduction is simply its length. Thus, for any λ-expression M, we define

$$\mathrm{Min}(M) = \begin{cases} \text{length of a minimum length reduction of } M \text{ to normal form,} \\ \quad \text{if } M \text{ has a normal form} \\ \infty, \text{ otherwise} \end{cases}$$

Now we discuss how to find the cost of evaluating an expression using an interpreter. There are various ways of implementing the λ-calculus. We can work in the usual framework of the λ-calculus; we can use graph representations of expressions; we can implement the substitution operation literally the way it is defined or simulate the cperation, *e.g.*, by using environments. It seems impossible to find uniform cost measure that works in all situations and thus, we hand-wave a little. First, we assume that any interpreter of the λ-calculus uses an operation that corresponds to reducing a β-redex. Second, we assume that the cost of this operation is about the same as the cost of reducing a β-redex in the usual framework of the λ-calculus. (Note that the actual cost of reducing a β-redex is not constant but depends on the size of the redex.) Third, we assume that if the interpreter performs other operations, then the cost of these operations can be subsumed in the cost of operation that is analogous to reducing a β-redex.

These assumptions are not too unreasonable. Consider Wadsworth's graph reduction technique. There is clearly a notion of reducing a redex; in fact, it is similar to the notion of reducing a redex in the λ-calculus. The technique also involves copying parts of a shared λ-abstraction. Copying, however, is done only before reducing a redex, and the combined cost of copying and reducing a redex is about the same as the cost of a β-reduction in the λ-calculus. Now consider environment based interpreters. These interpreter use a rule, usually called β rule, which is used to reduce a β-redex. However, the rule literally takes a constant amount of time as it does nothing but simply initiates the substitution. The actual substitution is carried out using other rules. However, the combined cost of applying the β rule and the other rules is again about the same as the cost of a β-reduction in the λ-calculus. In fact, these interpreters are more efficient because they can perform substitutions resulting from the reduction of more than one redex simultaneously.

With these assumptions in mind, we define the cost of reducing an expression M using an interpreter $\mathcal{I}$, written $\mathcal{I}(M)$, as follows:

$$\mathcal{I}(M) = \begin{cases} \text{number of operations analogous to reducing a } \beta \text{ redex performed,} \\ \quad \text{if the interpreter terminates} \\ \infty, \text{ otherwise} \end{cases}$$

Then,

An interpreter $\mathcal{I}$ is *optimal* if for any λ-expression M, $\mathcal{I}(M) \leq \text{Min}(M)$.

Some remarks: First, we, of course, don't require that an optimal interpreter, in reducing an expression, should produce a minimum length reduction beginning with expression. Furthermore, we don't insist that $\mathcal{I}(M)$ be strictly equal to Min(M). It seems unnecessary to do so, for all we care is that an optimal interpreter reduces an expression no less efficiently than a minimum length reduction beginning with the expression. We also believe that insisting on strict equality amounts to requiring that an optimal interpreter should produce minimum length reductions. Second, the definition implies that an optimal interpreter is a normalizing interpreter. Third, the definition admits the possibility that there are two interpreters, both of which are optimal according to our definition, but one is more efficient, in terms of the cost measure described above, than the other.

The above definition of an optimal interpreter is essentially the one that is implicit in [23, 24]. Vuillemin [38], and Berry and Lévy [8] use a similar definition of an optimal interpreter in the context of recursive program schemes. Barendregt *et al* [5] (see also [4]) defines the notion of an (L-1) optimal reduction strategy for the λ-calculus. The notion essentially gives us the following definition of an optimal interpreter: An interpreter is optimal if the number of β-redexes reduced by the interpreter in evaluating an expression is no more than the number of β-redexes reduced by *any other* interpreter in evaluating the expression. This definition of an optimal interpreter and the definition given earlier amount to the same thing in the usual setting of the λ-calculus for the following reason. The evaluation of an expression using an interpreter determines a reduction beginning with the expression. Conversely, for any reduction beginning with an expression, there is an interpreter that realizes it. If we admit interpreters that don't operate in the standard setting of the λ-calculus, then this definition is unsuitable because the class of all interpreters for the λ-calculus is not a well defined class.

1.2.5 Designing an optimal interpreter

This section is mostly a summary of some of the discussion in the previous sections.

As briefly noted in Section 1.1.2, designing an interpreter involves making appropriate choices for each of the following: representation of expressions, computation rule, and implementation of β rule of which the substitution operation is the most important part. To design an optimal interpreter, we have to find an appropriate combination.

We are, of course, interested in interpreters that use simple computation rules. Thus, "degenerate" interpreters like the one mentioned at the beginning of Section 1.2.2 are inadmissible.

As noted at the end of Section 1.2.2, there is no optimal interpreter among the interpreters that operate in the standard setting of the λ-calculus. The reason is that an interpreter in this setting is optimal only if it finds minimum length reductions, and we argued in Section 1.2.2 that there is no interpreter (excluding "degenerate" ones) that can do so.

Barendregt *et al* [5] (see also [4]) formally proved that there is no recursive (L-1) optimal reduction strategy. Rephrasing the result, there is no recursive computation rule that results in an optimal interpreter.

We strengthen the above result and show that even certain approximations to optimal computation rule don't exist. Specifically, we prove that, for *any* constant k, there is no recursive computation rule satisfying the property that if we reduce any λ-expression M using the computation rule, then the length of the generated reduction is no more than $k \times \text{Min}(M)$. See Appendix.

Note that the result says something about our assumption that the cost of a reduction is proportional to its length. It says that the exact value of the constant of proportionality is immaterial; one can choose it to be as large as one wishes.

At the end of Section 1.2.3, we raised the possibility of designing an optimal interpreter using a graph representation of expressions. We also said it is not obvious what the representation should be. Lévy gave a characterization of the kind of sharing of subexpressions that must be performed by an optimal interpreter. We describe Lévy's theory of optimal reductions in Chapter 2.

The problem, of course, is to find a graph representation of expressions that correctly implements Lévy's theory. In this thesis, we present such a representation.

1.3 Basic techniques for implementing λ-calculus

In the literature, we can identify three basic models for evaluating λ-expressions:

1. Literal substitution model.

2. Environment model.

3. Suspended substitution or functions on environments model.

In the first model, the β rule is implemented literally the way it is defined: substitute the argument part of the redex for all occurrences of the bound variable. Efficient implementation of the substitution operation, however, presents some difficulties. First, the substitution operation is a "large-grain" operation in the sense that it looks at the whole expression in which substitution is being made. Second, it involves renaming of bound variables to avoid capture of free variables. Renaming of variables requires recursive use of the substitution operation making its efficient implementation even more difficult.

In the other two models, the substitution operation, which is the main part of the β rule, is decomposed into smaller operations.

The environment model of evaluating expressions is well known. Classic examples of environment-based interpreters are Lisp interpreter and Landin's SECD machine [21], both of which use applicative order of evaluation. Henderson and Morris' lazy evaluator [15] is an environment-based interpreter that reduce redexes in leftmost order. Two features of these interpreters are worth noting. First, they implement lexical scoping of variables by using *closures*,*i.e.*, expression-environment pairs. Second, they don't evaluate expressions to normal form because they don't reduce redexes inside λ-abstractions. The main difficulty in reducing a redex inside a λ-abstraction is that it is not clear what the value of the variable bound by the λ-abstraction should be.

The basic idea underlying the third model is this. We introduce a new type of expression to carry out substitutions. To the usual definition of λ-expression, we add a new clause saying that an expression can also be of the form $\sigma\ x(M,N)$ where x is a variable and M,N are expressions. An expression of the form $\sigma\ x(M,N)$ represents the λ-expression that is obtained by substituting N for free occurrences of x in M. The β rule becomes quite simple.

(β): $\quad (\lambda x.M)\,N \longrightarrow \sigma x(M,N)$

σ-expressions are, then, propagated towards the leaves (*i.e.*, variable occurrences) by using rules that resemble various clauses of the substitution operation.

There are two different ways to view the third model. The first view is that the various clauses in the definition of the substitution operation has been made into reduction rules at

the same level as the β rule. The other view, which is more appealing to us, is that the model is like environment model except that we only specify *changes* to environments. Rather than working directly with environments we work with *functions on environments*. The main advantage of this model over literal substitution model is that performing substitutions, *i.e.*, propagating σ expressions, can be mixed with applications of the β rule. The main advantage over environment model is that expressions can be reduced to normal form. Examples of this model can be found in Staples [30] and Kennaway and Sleep [29].

As we noted earlier, renaming of bound variables also presents difficulties in implementing the substitution operation efficiently. The solution is to simply get rid of variable names. The issue of whether to work with variable names or use a name-free notation is largely independent of which of the above mentioned models to use.

De Bruijn proposed a name-free notation for λ-expressions in which an occurrence of a variable is coded as a number denoting the number of λ-abstractions between the occurrence and its binding λ-abstraction [12]. The β rule still involves "substituting" the argument part of the redex at appropriate places; however, it also involves adjusting some of the numbers. Number that need adjustment are "free" occurrences in the body of the abstraction part of the redex and in various copies of the argument part. In general, adjustments of numbers in different copies of the argument are different. DeBruijn's notation forms the basis of our interpreter and will be discussed in detail in the next chapter.

Curien's *weak* and *strong categorical combinatory logics*, though derived from the connection between the semantics of the λ-calculus and category theory, can be viewed as examples of environment model and functions on environment model in a name-free notation [11]. Indeed, we think they are the most elegant formulation of these models. The starting point for both calculii is DeBruijn's notation. Expressions in DeBruijn's notation are translated in expressions that *are* functions on environments using a set of primitive functions, *e.g.*, Apply, currying, pairing, and function composition. The weak calculus provides a set of rules that describe the applicative behavior of primitive functions and function composition. An expression is evaluated by simply applying it to an environment, usually an empty environment. The strong calculus provides a set of rules that describe the compositional behavior of primitive functions. These rules can be used to reduce expressions.

Graph reduction interpreters for the λ-calculus can be based on any of these models. Representative examples for the three basic models are: Wadsworth's interpreter for literal substi-

tution model, Henderson and Morris's lazy evaluator [15] for environment model, and Staples's interpreter [30] for the suspended substitution model.

None of the above interpreters, however, are optimal. Moreover, it seems that the three basic models are inherently unsuited for designing an optimal interpreter. The interpreter we propose combines ideas from literal substitution model and the functions on environment model.

1.4 Organization of the thesis

Chapter 2 describes Lévy's theory of optimal reductions and discusses difficulties in implementing his theoretical specification of an optimal reduction strategy for the λ-calculus.

Chapter 3 presents the main ideas underlying our interpreter, that is, the idea of sharing contexts as opposed to sharing of only subexpressions. We develop an interpreter for the λ-calculus that does sharing of contexts and describe the graph representation and the reduction rules used by the interpreter. At the end of the chapter, we discuss some of its deficiencies and suggest a number of improvements.

Chapter 4 presents the improved interpreter, which uses De Bruijn's notation for λ-calculus.

Chapter 5 presents the λ_f calculus. The calculus is used to relate reduction of graphs to reduction of λ-expressions and step in the proving the correctness of the interpreter. The calculus shows how changing of numbers that is part of the β-rule in De Bruijn's notation can be done in delayed fashion using environment-like structures. Although the calculus is motivated by the interpreter, it may be of interest in its own right.

Chapter 6 presents proofs of correctness and optimality.

Chapter 7 presents our conclusions and some ideas for further work.

Appendix presents the proof of the extension of Barendregt's result.

Chapter 2

Lévy's theory and its implications

In this chapter, we describe Lévy's theory of optimal reductions [23, 24] and discuss difficulties in implementing his theoretical specification of an optimal interpreter for the λ-calculus.

2.1 Overview of Lévy's theory

The goal is to design an optimal interpreter using the general strategy suggested in Section 1.2.3; that is, to design an optimal interpreter using a simple normalizing computation rule, *e.g.*, the leftmost rule, and a graph representation of expressions.

There are two ways to describe evaluation of expressions using a graph representation of expressions. The first way is to work explicitly with the graph representation. The second way is to model the effect of using the graph representation in terms of certain reductions in the expression world. Lévy used the second approach to give a theoretical specification of an optimal interpreter.

To model computations performed by graph reduction interpreters, we consider reductions over λ-expressions in which a *set* of redexes is reduced at each step. Such reductions are called *parallel reductions*. It is important to note that λ-expressions and parallel reductions form a system different from the λ-calculus; the first system is being used to describe certain types of interpreters for the second. Parallel reductions, however, are consistent with the usual (one-step) reductions in the λ-calculus; each step of a parallel reduction can be simulated by one or more steps in the λ-calculus. In this chapter, all reductions, unless indicated otherwise, are parallel reductions. So we use the same notation (*i.e.*, $\longrightarrow, \twoheadrightarrow$) for parallel reductions.

The main idea underlying the use of a graph representation is that of argument-sharing, and we want to consider only those interpreters that are based on this idea. Thus, only those

redexes that can be shared using argument-sharing should be reduced in one step. In general, reductions constructed according to Lévy's theory satisfy the condition.

Now, we illustrate the main idea underlying Lévy's theory using an example.

Example 2.1 Consider the expression

$$M \equiv D\,(F\,I) \quad \text{where } D \equiv \lambda x.x\;x, F \equiv \lambda x.x\;a, \text{ and } I \equiv \lambda x.x$$

The reduction graph of M is shown in Figure 2.1. In the figure, the reduction on the left hand side is the leftmost reduction of M to normal form, and the reduction on the right hand side is the minimum length reduction of M to normal form.

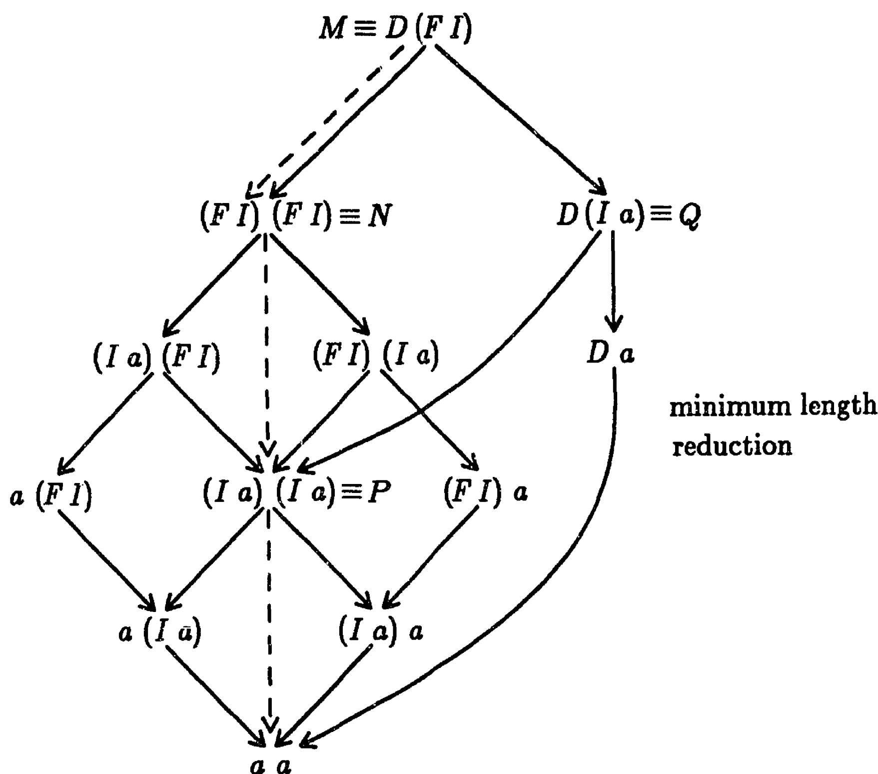


Figure 2.1: Reduction graph of M in Example 2.1

We assume that the initial expression M is represented as a tree. Thus, no redex in M is shared with another redex in M. Then, the first step is clear—we select the leftmost redex in M and reduce it.

$$\begin{array}{rcl} M & \equiv & \underline{D\,(F\,I)} \\ & \longrightarrow & (F\,I)\,(F\,I) \equiv N \end{array}$$

Now consider the next step. We, of course, have to reduce the leftmost redex in N. But we may also have to reduce some other redexes, the point being that a redex in a graph represents a set of redexes in the expression represented by the graph. So, what other redexes in N should be reduced at this step?

The strategy we use to answer the above question is this. We try to find as much as possible about the structure of a minimum length reduction beginning with M using the reduction that has been constructed so far, then use that information to decide what redexes should be reduced at the next step.

Continuing with our example, we note that the reduction of the leftmost redex in M creates two copies of the redex $F\,I$. But we can avoid copying of $F\,I$ by reducing it first. Thus, it is possible that a minimum length reduction beginning with M is of the following form, which is actually true for M (see Figure 2.1):

$$\begin{aligned} M &\equiv D\,\underline{(F\,I)} \\ &\longrightarrow D\,(I\,a) \longrightarrow \cdots \end{aligned}$$

On the other hand, it is also possible that no minimum length reduction beginning with M is of the above form, and $F\,I$ does get copied in a minimum length reduction beginning with M. But we have no effective way to decide which is the case, and thus, we simply assume that a minimum length reduction beginning with M is of the above form. Therefore, if we want to reduce M to normal form as efficiently as a minimum length reduction beginning with M, we should reduce the two redexes in N in one step. We have the following reduction:

$$\begin{aligned} M &\equiv \underline{D\,(F\,I)} \\ &\longrightarrow \underline{(F\,I)}\,\underline{(F\,I)} \equiv N \\ &\longrightarrow (I\,a)\,(I\,a) \equiv P \end{aligned}$$

Should the two redexes in P be reduced in one step? Again, we try to find as much as possible about a minimum length reduction beginning with M. Given the above reduction of M to P, we observe that there is another way to reduce M to P that can be obtained, in some sense, by interchanging the two steps. We first reduce $F\,I$ and then reduce $D \cdots$. That reduction is as follows:

$$\begin{aligned} M &\equiv D\,\underline{(F\,I)} \\ &\longrightarrow \underline{D\,(I\,a)} \equiv Q \\ &\longrightarrow (I\,a)\,(I\,a) \equiv P \end{aligned}$$

The last step in this reduction creates two copies of $I\,a$. Arguing as before, we can conclude that the two redexes in P should be done in one step.

Thus, we have the following reduction of M to normal form:

$$\begin{array}{rcl} M & \equiv & \underline{D\,(F\,I)} \\ & \longrightarrow & \underline{(F\,I)}\,\underline{(F\,I)} \equiv N \\ & \longrightarrow & \underline{(I\,a)}\,\underline{(I\,a)} \equiv P \\ & \longrightarrow & a\,a \end{array}$$

This reduction is shown using dashed arrows in Figure 2.1.

The number of steps in the above reduction is no more than the number of steps in the minimum length reduction of M to normal form. Moreover, redexes reduced at each step can indeed be shared using the basic idea of argument-sharing. For example, the reduction constructed above actually corresponds to the reduction of M to normal form using Wadsworth's graph reduction technique in conjunction with the leftmost computation rule.

Lévy's specified the set of redexes that must be reduced at a step in two different but equivalent ways. Consider the reduction of M constructed above. In the λ-calculus terminology, the two redexes in N are *residuals* of the redex $F\,I$ in M relative to the reduction of $M \longrightarrow N$. Thus, we can say that if we have constructed the reduction $S \twoheadrightarrow T$, then all redexes in T that are residuals (relative to this reduction) of a redex in a previous expression should be reduced in one step. This is true in Lévy's theory. On the other hand, consider the two redexes in P. They are not residuals of a redex relative to the reduction of M to P. The notion of residuals can be extended to cover such cases; see [24].

The simplest, though somewhat mechanical, way to keep track of redexes that should be reduced in one step is to use a scheme to *label* subexpressions. The labelling scheme should be such that redexes that should be reduced in one step have the same "identifying" label. Lévy's labelled λ-calculus, which we discuss next, provides such a scheme.

2.2 Lévy's labelled λ-calculus

In this section, we present Lévy's labelled λ-calculus ([22, 23]). (See also [19] and [4], which present a slightly different version.)

Definition 2.2 (Labels) Let $\mathbf{L}_0 = \{a, b, \ldots\}$ be an infinite set of of symbols, called *elementary* labels. The set $\mathbf{L}$ of (Lévy) labels is defined inductively as follows:

1. $w \in \mathbf{L}_0 \Rightarrow w \in \mathbf{L}$

2. $w \in \mathbf{L} \Rightarrow \underline{w} \in \mathbf{L}$.

3. $w \in \mathbf{L} \Rightarrow \overline{w} \in \mathbf{L}$.

4. $w, v \in \mathbf{L} \Rightarrow wv \in \mathbf{L}$.

Here wv is the concatenation of words w and v. Labels will be denoted by $u, v, w, \ldots$.

Given a set of elementary labels $\{a, b, c\}$, some examples of labels are a, abc, $a\underline{b}c$, $\underline{a\overline{b}c}$. Note that labels may have any level of nested underlinings.

Definition 2.3 (Labelled λ-terms) Let $\mathbf{V}$ be an infinite set of variables. Terms of the labelled λ-calculus, Λ_L, are given by the following grammar:

$$\begin{array}{rcl} Term & := & x \in \mathbf{V} \mid Term\,Term \mid \lambda x.Term \\ & & \mid Term^w \text{ where } w \in \mathbf{L} \end{array}$$

The first three clauses generate the usual λ-terms. Thus, we have defined terms with "partial labelling", ie, not every subterm carries a label. An important type of labelled terms are the following:

Definition 2.4 A labelled term is called *elementarily labelled* if every subterm of the term carries an atomic label.

We define two reduction rules on labelled terms.

Definition 2.5 (Reduction rules)

(lab): $\quad (M^w)^v \longrightarrow M^{wv}$

(β_L): $\quad (\lambda x.M)^w\, N \longrightarrow M[x := (N)^{\underline{w}}]^{\overline{w}}$

That is, each free occurrence of x in M is replaced by $N^{\underline{w}}$, and the result is labelled by $\overline{w}$. Note that N may carry some labels itself. The necessary renaming of variables during substitution is done in a way such that labels don't get affected. We will not pay much attention to *lab* rule, and assume that this rule is applied as soon as possible.

Example 2.6 Here is an example of a labelled reduction; the initial expression is a labelled version of Dy.

$$
\begin{aligned}
M_L &\equiv ((\lambda x.(x^a\, x^b)^c)^d\, y^e)^f \\
&\longrightarrow (y^{e\underline{d}a} y^{e\underline{d}b})^{c\overline{d}f}
\end{aligned}
$$

There is an obvious correspondence between reductions in the λ-calculus and the reductions in the labelled calculus. If M_L is a labelled term, then we write $erase(M)$ for the λ-term obtained by erasing all labels. If $\sigma_L : M_L \twoheadrightarrow N_L$ is a labelled reduction, then there is also a corresponding reduction in the λ-calculus, which is obtained by erasing all labels; we denote this reduction by $erase(\sigma_L)$. On the other hand, if $\sigma : M \twoheadrightarrow N$ is a reduction in the λ-calculus, then we can "lift" this reduction to a labelled reductions by considering a labelling of M.

2.3 Optimal reductions

In this section, we define optimal reductions in the λ-calculus using the labelled calculus. We need the following two definitions.

Definition 2.7 The *degree* of a redex is the label of its abstraction part.

Definition 2.8 Let

$$\sigma : M \equiv M_0 \xrightarrow{\mathbf{R_0}} \cdots M_i \xrightarrow{\mathbf{R_i}} M_{i+1} \cdots \xrightarrow{\mathbf{R_n}} M_{n+1}$$

be a labelled reduction where $\mathbf{R}_i$ is a set of redexes in M_i.

1. σ is called a *one-degree* reduction if all redexes in $\mathbf{R_i}$, for $0 \leq i \leq n$, have the same degree.

2. σ is called a *complete* reduction if the set $\mathbf{R_i}$, for $0 \leq i \leq n$, includes all and only those redexes in M_i that have the same degree.

3. σ is called a *leftmost complete* reduction if it is a complete reduction, and furthermore, the set $\mathbf{R_i}$, for $0 \leq i \leq n$, includes the leftmost redex in M_i.

The following theorem characterizes the optimal reductions in the λ-calculus.

Theorem 2.9 *Suppose M is a λ-term that has a normal form. Let M_L be an elementarily labelled term such that $erase(M_L) = M$. Let σ_L be the leftmost complete reduction of M_L to normal form. Then, $erase(\sigma_L)$ is an optimal reduction of M to normal form.*

The proof of the theorem relies on the next two propositions.[1]

[1] Proofs of these or similar propositions appear in Lévy's thesis. In [8], similar propositions are proved for recursive program schemes.

Proposition 2.10 *In a complete reduction starting with an elementarily labelled term, no two distinct steps reduce redexes with the same degree.*

Proposition 2.11 *Suppose M is an elementarily labelled term, and furthermore, suppose that M has a normal form. Then, the leftmost complete reduction of M to normal form has the minimum length among all one-degree reductions of M to normal form.*

Proof of Theorem 2.9. Let τ be a minimum-length ordinary reduction (*i.e.*, not parallel) reduction of M to normal form. Lift τ to a labelled reduction τ_L of M_L to normal form. Now, τ_L reduces exactly one redex at each step, so it is a one-degree reduction. By Proposition 2.11, $|\sigma_L| \leq |\tau_L|$. Now, $|\tau| = |\tau_L|$, and $|\sigma_L| = |erase(\sigma_L)|$. Thus $erase(\sigma_L)$ is an optimal reduction of M to normal form. □

The following proposition will be useful in proving the optimality of the interpreter.

Proposition 2.12 *Let $M_L \twoheadrightarrow N_L$ be a complete reduction starting with an elementarily labelled term. Let w be the degree of redex inside N_L. Then no label inside N_L contains either $\overline{w}$ or $\underline{w}$.*

2.4 Difficulties in implementing Lévy's theory

The problem, of course, is to find a representation of λ-expressions using which optimal reductions as specified by Lévy' theory can be implemented. And obviously, the representation should be such that each step of an optimal reduction can be carried out in a cost that is approximately the same as the cost of a β-reduction in the λ-calculus. In this section, we discuss difficulties in designing such a representation.

All reductions in this sections are reductions constructed according to Lévy's theory. Moreover, $I \equiv \lambda x.x$ and $D \equiv \lambda x.xx$.

2.4.1 Properties of redexes to be shared

We illustrate certain properties of redexes to be shared. In a reduction constructed according to Lévy's theory, redexes reduced in a step

1. may not be identical subexpressions, and

2. may not be disjoint subexpressions.

The following two examples illustrate these points.

Example 2.13 Consider the following reduction:

$$\begin{array}{rcl} M & \equiv & \underline{(\lambda x.x\,a\,(x\,b))\,(\lambda y.I\,y)} \\ & \longrightarrow & \underline{(\lambda y.I\,y)\,a}\,((\lambda y.I\,y)\,b) \\ & \longrightarrow & \underline{I\,a}\,((\lambda y.\underline{I\,y})\,b) \\ & \longrightarrow & a\,((\lambda y.y)\,b) \end{array}$$

The redexes reduced in the last step, namely $I\,a$ and $I\,b$ are *not identical* subexpressions.

Example 2.14 Now, Consider the following reduction:

$$\begin{array}{rcl} M & \equiv & \underline{(\lambda x.x\,x)\,(\lambda y.(\lambda z.y\,z)\,a)} \\ & \longrightarrow & \underline{(\lambda y.(\lambda z.y\,z)\,a)\,(\lambda y.(\lambda z.y\,z)\,a)} \\ & \longrightarrow & \underline{(\lambda z.(\lambda y.\underline{(\lambda z.y\,z)\,a})\,z)\,a} \end{array}$$

The two redexes reduced in the last step are *not disjoint* subexpressions.

As we noted in the last section, residuals of a redex are to be shared, and both these properties are essentially properties of residuals of a redex. They come from the presence of bound variables in the λ-calculus.[2] A redex may contain free occurrences of a bound variable, and the substitutions performed for these free occurrences of the variable may be different in different residuals of the redex. In Example 2.13, y occurs free in the redex $I\,y$. At the second step of the reduction, the free occurrence of y in the leftmost residual gets substituted by 3 whereas the free occurrence of y in the other residual does not.

Thus, the representation of expressions should permit the sharing of subexpressions that, in some sense, differ in substitutions for bound variables. This suggests that we should look at implementation schemes in which substitution is suspended, *e.g.*, using environments.

2.4.2 Sharing and copying of applications

Redexes do get created during the reduction, and it is applications that become redexes. To share redexes, it is necessary to share applications which are not redexes as they may become redexes to be shared.

[2]Residuals of a redex in rewriting systems like the SK-calculus and recursive program schemes are always syntactically identical and disjoint, which in part explains why directed acyclic graph representations of expressions suffice to design optimal evaluators for these systems.

Example 2.15 Consider the optimal reduction constructed at the beginning of this chapter, which is reproduced below.

$$\begin{array}{rcl} M & \equiv & \underline{D\,((\lambda x.x\,a)\,I)} \\ & \longrightarrow & \underline{(\lambda x.x\,a)\,I}\,\underline{((\lambda x.x\,a)\,I)} \;\; \equiv N \\ & \longrightarrow & \underline{I\,a}\,\underline{(I\,a)} \\ & \longrightarrow & a\,a \end{array}$$

The two occurrences of application $x\,a$ in N become redexes that are to be shared. Thus, it is necessary that N be represented in a way so that these applications are shared. These applications, in some sense, come from the single application $x\,a$ in M. In the λ-calculus terminology, they are *descendants* of $x\,a$ in M.

Consider, on the other hand, the following example.

Example 2.16 Let

$$M \equiv ((\lambda x.x\,I(x\,J))\,(\lambda y.I\,(y\,a)) \text{ where } J \equiv \lambda x.a$$

First few steps in the optimal reduction of M are as follows:

$$\begin{array}{rcl} M & \equiv & \underline{((\lambda x.x\,I(x\,J))\,(\lambda y.I\,(y\,a))} \\ & \longrightarrow & \underline{(\lambda y.I\,(y\,a))\,I}((\lambda y.I\,(y\,a))\,J) \equiv N \\ & \longrightarrow & I\,(I\,a)((\lambda y.I\,(y\,a))\,J) \end{array}$$

Now, consider the two occurrences of ya in N. At the last step, the leftmost occurrence becomes a redex whereas the other doesn't. In the last expression, the redex $I\,a$ certainly should not be shared with the application $y\,a$ — one is a redex whereas the other is not even a redex. Note that two occurrences of $y\,a$ in N are descendants of a single application in M.

Comparing these two examples, descendants of an application in the initial expression should be shared as long as there is a possibility that they may become redexes that are to be shared (as in Example 2.15). However, once some of the descendants of an application become redexes, they should not be shared with other descendants (as in Example 2.16). Thus, certain amount of *copying* of shared subexpressions is necessary.

Sharing descendants of an application presents all the difficulties mentioned in the last section because descendants of an application also have both properties. However, now there is the added complication due to copying of shared subexpressions. Consider Example 2.16. The two redexes $I\,(I\,a)$ and $I\,(y\,a)$ are to be shared. They are certainly not identical; they, in

some sense, differ in the substitutions for y. Now consider the fact that $I\,a$ and $y\,a$ are not to be shared. What this implies is that we not only need to share subexpressions that differ in substitutions for variables but also need to share subexpressions that differ more generally. In Chapter 3, we shall argue that we need a representation that permits sharing of *contexts.*

We emphasize that both sharing of subexpressions and copying of shared subexpressions are necessary. Sharing of subexpressions is necessary to get an *optimal* interpreter whereas copying of shared subexpressions is necessary to get a *correct* interpreter.

2.4.3 Avoiding capture of free variables

The substitution operation of the λ-calculus involves renaming of bound variables to avoid capture of free variables. In this section, we discuss difficulties in carrying out the required renaming of bound variables.

Example 2.17 Consider the following reduction:

$$\begin{aligned} M &\equiv \underline{(\lambda x.x\,x)\,(\lambda y.a\,(\lambda z.y\,(I\,z)))} \\ &\longrightarrow \underline{(\lambda y.a\,(\lambda z.y\,(I\,z)))\,(\lambda y.\,(\lambda z.y\,(I\,z)))} \\ &\longrightarrow a\,(\lambda z.\underline{(\lambda y.a\,(\lambda z.y\,(I\,z)))\,(I\,z)}) \qquad \equiv N \end{aligned}$$

Both the outermost and the innermost λ-abstractions in N bind same variable z. To reduce the underlined redex, it is necessary to rename variable bound by the innermost λ-abstraction. Note that the variable bound by the outermost λ-abstraction should not be renamed. Actually, what we must ensure is that the variables bound by the outermost and innermost λ-abstraction are named differently. So suppose we rename z to v for the innermost λ-abstraction. Then, the next step of the reduction is as follows:

$$N \longrightarrow a\,(\lambda z.a\,(\lambda v.\underline{I\,z}\,(\underline{I\,v})))$$

The two underlined redexes are to be shared, and we see that they are not identical.

Again, we find that we need a representation that permits sharing of subexpressions that are not identical. This is to be expected, for the renaming of bound variables is a very simple form of the substitution operation.

There is, however, an additional complication. Consider the expression N. The two λ-abstractions with z as bound variable come from the same λ-abstraction in the initial expression M—they are descendants of a λ-abstraction in M. These abstractions are arguments of two

applications, namely application of a to something, which themselves are descendants of an application in M. In general, these applications may have to be shared. Now suppose we start with the basic idea of sharing arguments using a graph representation of expressions. Then, it is reasonable to assume that if two applications are shared, then their argument parts are also shared. Thus, the representation of N would look like something like the graph in Figure 2.2.

Of course, it is not clear how to interpret the graph in Figure 2.2 as the expression N, but

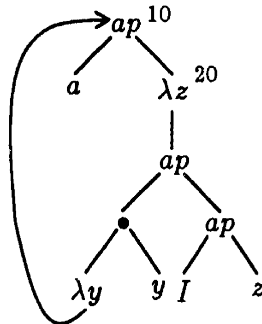


Figure 2.2: Schematic representation of N to illustrate copying of λ-nodes

that is not the point. The important point is that the applications with a as function part are represented by a single node, Node 10. Similarly, the λ-abstractions with z as bound variable are represented by a single node, Node 20. Now consider the fact that we wish to rename the bound variable of only one of the λ-abstraction represented by Node 20. To do so, it is necessary that we create two copies of Node 20. It is not clear how to copy Node 20 without copying Node 10, and we certainly don't want to create two copies of Node 10.

We believe that renaming of bound variables is not a viable option for avoiding capture of free variables. We have to use some other mechanism. Environment based interpreters, for example, use closures for this purpose.

2.5 Summary

In this chapter, we described Lévy's specification of an optimal reduction strategy for the λ-calculus. We also discussed the main difficulties in implementing his specification.

Chapter 3

An Optimal Interpreter for the λ-calculus

The basic idea underlying our interpreter is a new graph representation of expressions that permits sharing of not only sub-expressions but also *contexts*,*i.e.*, parts of an expression that are not complete sub-expressions. First, we motivate the idea of sharing contexts. Then, we develop a preliminary version of our interpreter.

The emphasis in this chapter is on convincing the reader that the ideas presented are sound and that they lead to a correct and optimal interpreter. Although the interpreter developed in this chapter satisfies the requirement of being an optimal interpreter, it is not quite satisfactory. In Section 3.9, we discuss some of its deficiencies and suggest a number of improvements. The last section describes one of the suggested improvements. In Chapter 4, we will incorporate other suggestions by switching to De Bruijn's notation for the λ-calculus.

3.1 The problem with (ordinary) graph reduction

In Section 1.2.3, we described a way to share arguments using a graph representation of expressions. The basic idea presented there, however, needs to be refined. The problem, which we explain using an example, comes from shared λ-abstractions.

Example 3.1 Consider the following expression, which is the same expression as in Example 2.16.

$$M \equiv ((\lambda x.x\, I(x\, J))\, (\lambda y.I\, (y\, a)), \text{ where}$$
$$I \equiv \lambda x.x \text{ and } J \equiv \lambda x.a$$

The reduction of the leftmost redex in M is easy. Figure 3.1(a) shows the graph after the reduction.

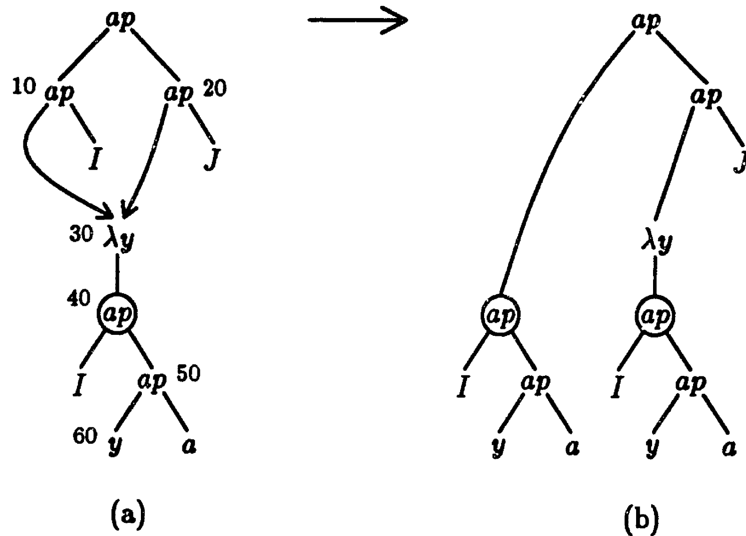


Figure 3.1: (a) Graph after the reduction of the leftmost redex in M (b) Graph after copying the shared abstraction and reducing the leftmost redex

Now consider the reduction of the leftmost redex (*i.e.*, redex represented by Node 10) in this graph. The reduction involves substitution of I for occurrences of bound variable y. But, the abstraction part of the leftmost redex is represented by the subgraph enclosed in the triangle, which also represents the abstraction part of another redex, *i.e.*, redex represented by Node 20. So substituting I for y directly gives wrong result, for the substitution should not affect the abstraction part of the other redex.

An obvious solution to the problem is to create two copies of the shared abstraction. Figure 3.1(b) shows the graph after copying the shared abstraction and reducing the leftmost redex. Wadsworth [39] observed that copying all of a shared abstraction is unnecessary and gave a rule for copying only parts of a shared abstraction. Arvind *et al* [1] describe a copying rule that models lazy interpreter of Henderson and Morris.

Indiscriminate copying, however, may result in an interpreter that is not optimal for the simple reason that copying destroyes sharing. For example, an interpreter that copies Node 40 cannot be optimal. Intuitively, Node 40 represents two copies of a single redex $I\ (y\ a)$ in the initial expression M. An interpreter that matches the performance of a minimum length reduction of M to normal form must reduce both copies in one step. Also as noted in Example 2.16, Lévy's theory specifies that the two redexes represented by Node 40 should be reduced in

one step.

Not surprisingly, all the copying rules mentioned above result in interpreters that are not optimal—they all imply copying of Node 40 in the above example. [1]

3.2 The idea of sharing contexts

To reiterate the discussion in the last section, it is necessary to copy certain parts of a shared abstraction, and it is equally necessary *not* to copy certain other parts. Both sharing and copying are equally important in designing an optimal interpreter; sharing is necessary for optimality, and copying is necessary for correctness.

Copying, however, destroys sharing. Thus, we should do only as much as copying as is absolutely necessary to perform the next reduction. Consider a redex whose abstraction part is shared. The reduction of the redex is going to affect certain parts of the abstraction; the ones really affected are

1. the root of the abstraction, which is eliminated, and

2. occurrences of the bound variable, which are replaced by an expression.

Thus, we should copy *only* these parts of the abstraction to reduce the redex.

Consider, for example, the graph in Figure 3.1(a). To reduce the leftmost redex, we should copy only Node 30 (root of the abstraction) and Node 60 (occurrence of y). Then, we can use one copy of these nodes to reduce the leftmost redex. Figure 3.2(a) shows schematically the graph after copying of these nodes, and Figure 3.2(b) shows the graph after the reduction of the leftmost redex.

To explain the most important feature of graphs in Figure 3.2, we need the notion of *contexts*. A *context* is a λ-expression with a "hole"; or pictured as a tree, it is a tree with one missing subtree. The hole in a context is written as $\square$. Further, we write $C[]$ for a context and $C[A]$ for the λ-expression obtained by "filling" the hole in C by A. Note that free variables inside A are allowed to become bound in $C[A]$.

The subgraph that is still shared in Figure 3.2(a) is an example of a *shared context*. It represents two instances of the context $I\ (\square\ a)$ in the expression represented by the graph.

[1] On the other hand, we should point out that there are techniques for implementing λ-calculus that can reduce the two redexes represented by Node 40 in one step (see, for example, Staples [30]). None of these techniques, however, give us an optimal interpreter. See also Section 1.3.

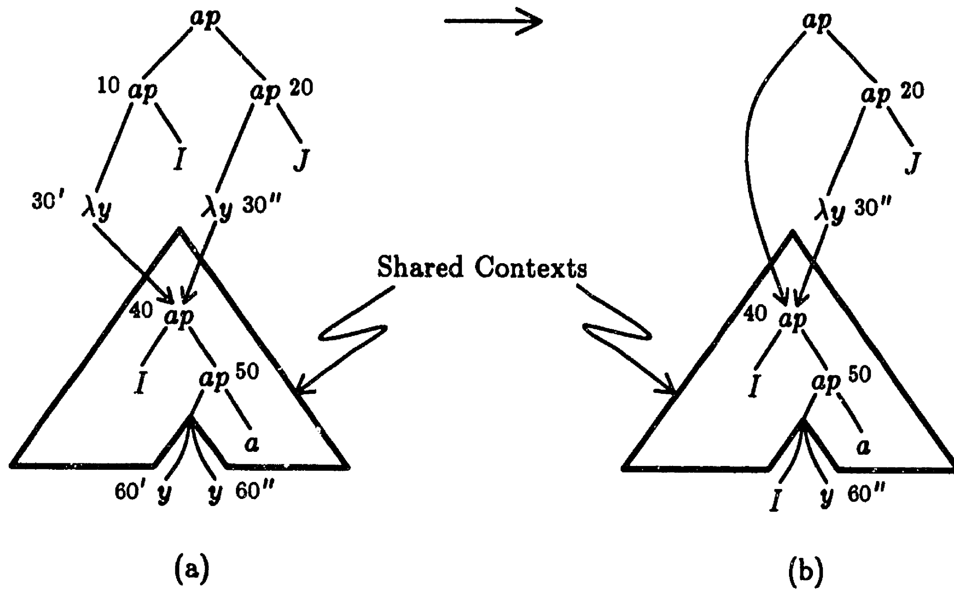


Figure 3.2: (a) Graph after copying of Nodes 30 and 60 (b) Graph after the reduction of the leftmost redex

The same is also true of the shared subgraph in Figure 3.2(b). The two graphs in Figure 3.2, however, show different situations in which a context is shared. In the first graph, holes in both instances of the context represented by the shared subgraph are filled with y. This is not the case in the second graph; the hole in one instance is filled with I (substitution for y) whereas the hole in the other instance is filled with y.

Clearly, it is not sufficient to copy only the root and occurrences of the bound variable; certain other parts of a shared λ-abstraction also may need to be copied. For example, Node 50 in Figure 3.2(a) should also be copied. Intuitively, Node 50 represents two applications. One of them will become the redex $I\ a$ after the reduction of the leftmost redex, whereas the other will become the redex $J\ a$ after the reduction of the redex represented by Node 20. It is unreasonable to say that these redexes be shared, since the result of reducing one redex will be quite different than the result of reducing the other redex. And as discussed in Section 2.4, Lévy's theory says that these redexes should not be done in one step.

Copying of Node 50 can be done by narrowing the boundaries of the shared context, *e.g.*, as shown in Figure 3.3. The figure shows a different situation in which a context is shared; holes in the two instances of the context are filled with neither variables nor substitutions for variables.

Copying of Node 50, however, is not necessary to reduce the leftmost redex. It is necessary

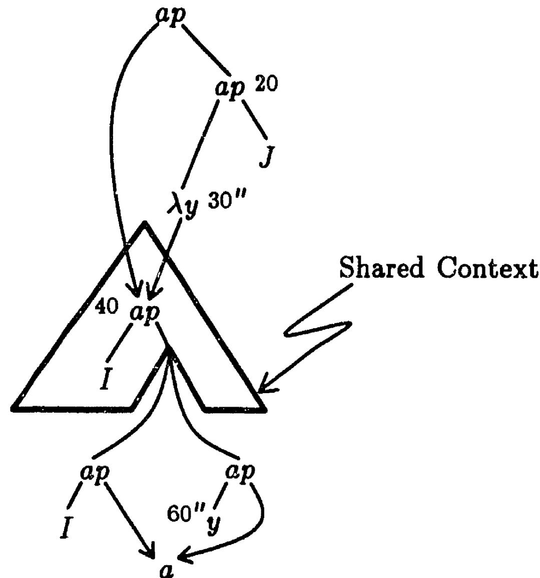


Figure 3.3: The graph after copying Node 50

only when reducing one of the redexes represented by Node 50 and thus should be done at that time. In a way, copying of the root and occurrences of the bound variables initiates copying of other parts; the actual copying can be done as and when needed.

Summarizing the discussion, we need a graph representation of expressions using which we can share not only sub-expressions but also contexts. Commonly used graph representations of expressions cannot share contexts; they can share only subexpressions.

3.3 A representation for sharing contexts

In this section, we develop a graph representation of expressions that permits sharing of contexts. We also give the translation of graphs into expressions. In the subsequent sections, we will use this representation, with a few extensions, to design a preliminary version of our interpreter.

We begin with a simple situation of a shared context. Suppose $M \equiv C[A]$ and $N \equiv C[B]$ where C is a context, and A and B are expressions, not necessarily identical. Our task is to represent M and N with only one copy of $C[]$.

Figure 3.4(a) shows schematically M and N. Nodes drawn as squares are "dummy" root nodes; they are used to refer to M and N in a convenient way.

To share the context $C[]$, we represent $C[]$ as usual. Rather than filling the hole in $C[]$ with

either A or B, we fill the hole with both using a new type of node, called a *conditional* node; see Figure 3.4(b). Intuitively, the subgraph rooted at Node 10 in Figure 3.4 is supposed to represent M. Similarly, the subgraph rooted at Node 20 is supposed to represent N.

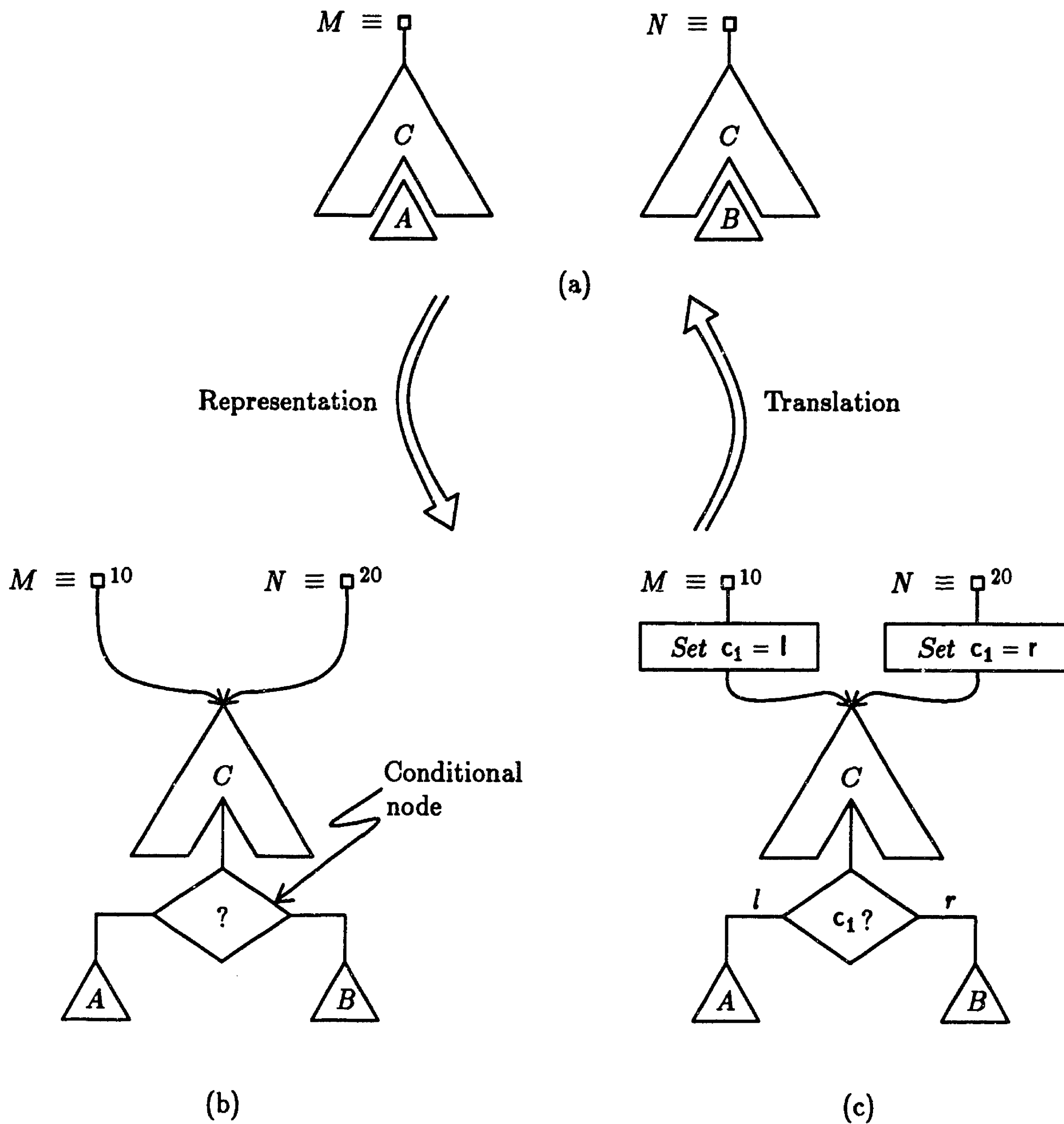


Figure 3.4: (a) Expressions M and N (b) The context $C[]$ with hole filled with a conditional node (c) The graph representing M and N with only one copy of $C[]$

A conditional node has three parts:

- Two subgraphs representing λ-expressions, which in this case are A and B.
- A condition, which we will discuss shortly.

In figures, we draw a conditional node as a diamond (like conditionals in a flow chart). A conditional node is interpreted as representing either one or the other expression but not both; the interpretation is governed by the condition part of the node.

To be precise about our intuitive understanding of the graph in Figure 3.4(b), we should give a procedure to translate graphs into λ-expressions. The procedure when applied to Node 10 should return M and when applied to Node 20 should return N. Thus, if the procedure starts with Node 10, it should interpret the conditional node as representing A, since $M \equiv C[A]$. Similarly, if the procedure starts with Node 20, it should interpret the conditional node as representing B, since $M \equiv C[B]$. Putting it another way, there are two pointers to the shared structure and there are two pointers out of the conditional node. If the procedure enters shared structure via the left pointer, it should exit the conditional node also via the left pointer. Similarly, if it enters the shared structure via the right pointer, it should exit the conditional node also via the right pointer. Thus, the translation procedure has to keep track of how it entered the shared structure in order to decide how it should exit the conditional node.

A simple way to do it is this. We associate a variable, say c_1, with the conditional node. The translation procedure maintains a value for c_1 while translating the graph. If it enters the shared structure via the left pointer, it sets c_1 to the value l (for left); if it enters via the right pointer, it sets c_1 to the value r (for right). At the conditional node, it simply uses the value of c_1 to decide which pointer to follow.

The graph in Figure 3.4(b), however, doesn't contain enough information for the translation procedure to work as described above. There is no information that associates c_1 to the conditional node or that describes how the value of c_1 should be set.

Thus, we add all the relevant information to the graph in Figure 3.4(b). Figure 3.4(c) shows the new graph. Variables, like c_1, will be called *control variables*, and we assume that their names don't conflict with names of λ-calculus variables. Nodes drawn as boxes are yet another new type of nodes. They contain phrases like "Set $c_1 = l$" or "Set $c_1 = r$", which direct the translation procedure to do what the phrase says. The conditional node now contains a condition, written as "c_1?", which directs the translation procedure to examine the value of c_1 for deciding which pointer to follow.

It is easy to see that if we apply the translation procedure as described earlier to Node 10 in Figure 3.4(c), then it returns M; and if we apply it to Node 20, then it returns N.

The situation we discussed so far was quite simple. In general, we will have more complex

situations of shared contexts, *e.g.*, a context shared among more than two expressions, a shared context embedded inside another shared context. All these situations can be handled easily by introducing as many conditional nodes, control variables, and box nodes as necessary. Figure 3.5 shows graphs for the two examples mentioned above.

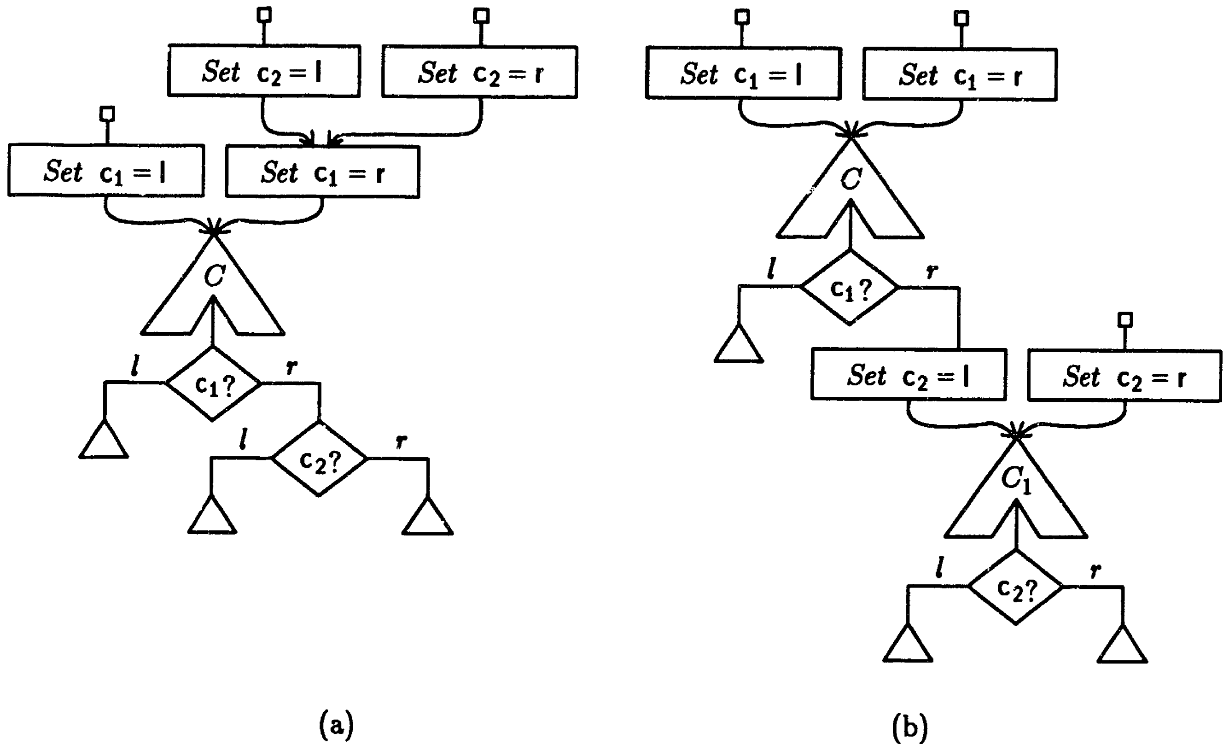


Figure 3.5: (a) A context shared among more than two expressions (b) A shared context embedded inside another shared context

The point is that, in general, the translation procedure has to maintain the values of several control variables while translating a graph. Thus, we introduce the notion of *control environments.* A control environment is a list of (variable, value) pairs. Variables are control variables $c_1, c_2 \ldots$, and values are either l or r,.

We define the translation procedure as a recursive procedure of two arguments. The first argument is the root node of the graph (or subgraph) to be translated, and the second argument is a control environment. The procedure returns the λ-expression represented by the graph (or subgraph) *relative* to the control environment. Thus,

$$Tr : \text{Graphs} \rightarrow \text{Control environments} \rightarrow \lambda\text{-expressions}$$

The expression represented by a graph is obtained by translating the graph relative to the empty environment ().

With this view of the translation procedure, phrases like "Set $c_1 = l$" tell the translation procedure to add a new (variable, value) pair to a control environment. Conditions like "c_1?" tell the translation procedure to look-up the value of a control variable in a control environment. Therefore, we define the following functions on control environments; ρ denotes a control environment, and *val* is either l or r.

$$\begin{aligned} Extend\, c_i\, val\, \rho &= ((c_i, val), \rho) \\ Value\, c_i\, \rho &= val \text{ where } val \text{ is the value associated with } c_i \text{ in } \rho \end{aligned}$$

Using these functions, the graph in Figure 3.4(b) becomes the one shown in Figure 3.6(a).

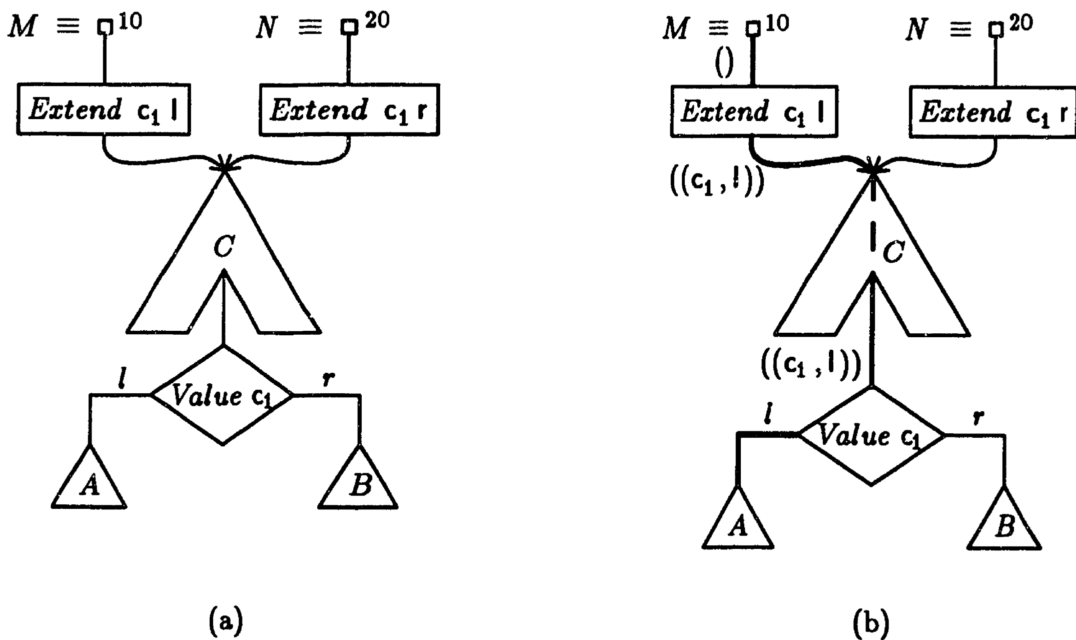


Figure 3.6: (a) Representation of M and N using functions on control environments (b) Illustration of the working of the translation procedure

The clauses in the definition of the translations procedure that are relevant to our discussion are as follows (N is the root node of a graph):

$$Tr\, N\, \rho \;=\; \textbf{case } N \textbf{ of}$$

...

$$\boxed{f} \rightarrow N_1 \;:\; Tr\, N_1\, (f\, \rho)$$

$$\langle f \rangle \rightarrow N_1, N_2 \;:\; \begin{cases} Tr\, N_1\, \rho & \text{if } f\, \rho = l \\ Tr\, N_2\, \rho & \text{if } f\, \rho = r \\ \text{"Error"} & \text{otherwise} \end{cases}$$

To illustrate the working of the translation procedure, we apply it to Node 10 in Figure 3.6(a). Figure 3.6(b) shows the path followed by the procedure as well as control environments at various points during the translation.

We are now done. We have a graph representation of expressions that permits sharing of contexts, and we know how to relate graphs to λ-expressions.

Before we leave this section, we would like to emphasize the distinction between the translation procedure and the interpreter, which we haven't discussed in this section. The translation procedure relates graphs to λ-expressions; the relationship is essential to show that the interpreter implements the λ-calculus correctly. On the other hand, the interpreter works only on graphs; it transforms a graph to another graph by applying certain reduction rules, *e.g.*, β-rule. The interpreter never translates a graph to an expression except perhaps at the end to print the result. There is nothing new in this distinction. It has to be made in describing any graph reduction interpreter, *e.g.*, Wadsworth's interpreter. Usually the translation is quite simple—it simply unravels a given graph. In our case, the procedure is quite complex, and we spent quite some time explaining it.[2]

3.4 Reducing a β-redex

In this section, we describe how the interpreter reduces a β-redex. The interesting case, of course, is the one in which the abstraction part of the redex is shared. The reduction is done in two steps: copying and actual reduction.

Figure 3.7(a) shows schematically a redex whose abstraction part is shared. As suggested in Section 3.2, we should copy only the root of the abstraction and occurrences of bound variables to reduce the redex. Copying of the root of the abstraction, Node 10, is not a problem. Copying of occurrence of x, Node 20, implies that two instances of the body of the abstraction be represented with one copy of the context $C[]$. To do so, we use the ideas described in the last section.

The resulting graph is shown in Figure 3.7(b). The transformation described by the pair of graphs in parts (a) and (b) will be called λ-*splitting*. It should be read as follows. The

[2] An analogy may help. A graph is like Tax Laws of U.S.A. Conditional nodes and box nodes are like conditional clauses in Tax Laws, *e.g.*, the one telling you when you can deduct your lunch as a business expense. The translation procedure is like a battery of tax-lawyers and consultants trying to unravel the mysteries of Tax Laws. Control environments are like notes of these lawyers to keep track of important points. So what is the interpreter like? It is like Congress and the Administration combined, always trying to simplify the laws. Well, come to think of it, the last analogy isn't quite right.

control variable c_i is a fresh control variable. On the right hand side, all nodes except the *ap* node and the ones inside subgraphs A and C are new nodes. The pointer from the *ap* node to Node 10 is redirected to Node 10'; all other pointers to the Node 10 are redirected to Node 10". Furthermore, all pointers to Node 20 (occurrence of x) are redirected to the conditional node.

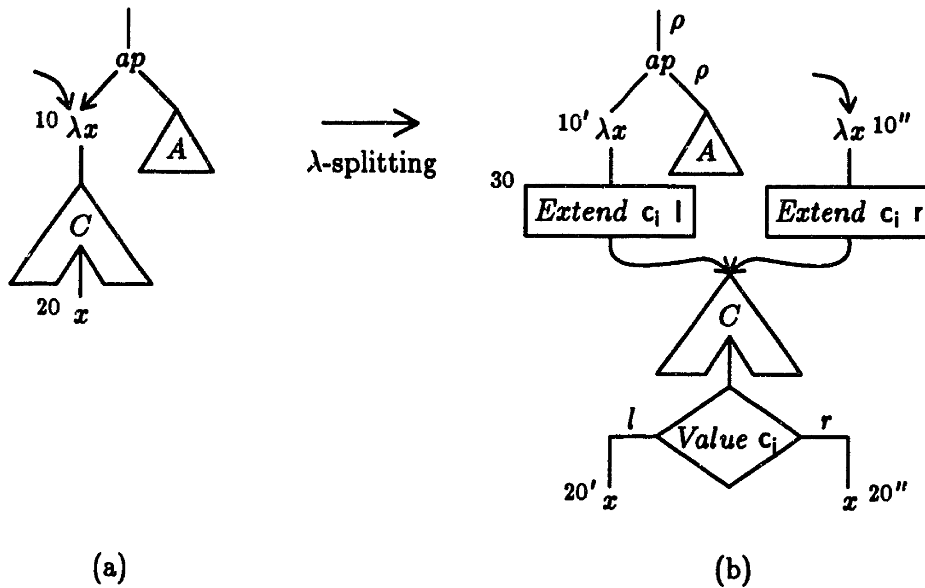


Figure 3.7: (a) A β-redex whose abstraction part is shared (b) Graph after copying the root of the abstraction and occurrence of bound variable. Transforming the graph in (a) to the one in (b) is called *λ-splitting.*

A couple of points: First, the transformation preserves translation. Suppose the translation of the subgraph rooted at Node 10 relative to ρ is $\lambda x.M$. Then, it is clear that the translation of the subgraph rooted at either Node $10'$ or Node $10''$ is also $\lambda x.M$. Second, the relationship between the graph in part (b) and its translation implies that x represented by Node $20'$ is bound by the λ-abstraction represented by Node $10'$. The same is also true for Nodes $10''$ and $20''$. Putting it differently, the procedure to find all occurrences of a bound variable is similar to the translation procedure.

The second step is to actually reduce the redex. The obvious way to do this is to replace occurrences of the variable bound by the λ-abstraction by pointers to the argument part of the redex. Figure 3.8 shows the graph after this operation. Notice that the pointer to Node 20' is redirected to A and all pointers to the *ap* node are redirected to Node 30 (the result of the reduction).

Is this obvious way of reducing a redex correct? The criterion for correctness, of course, is

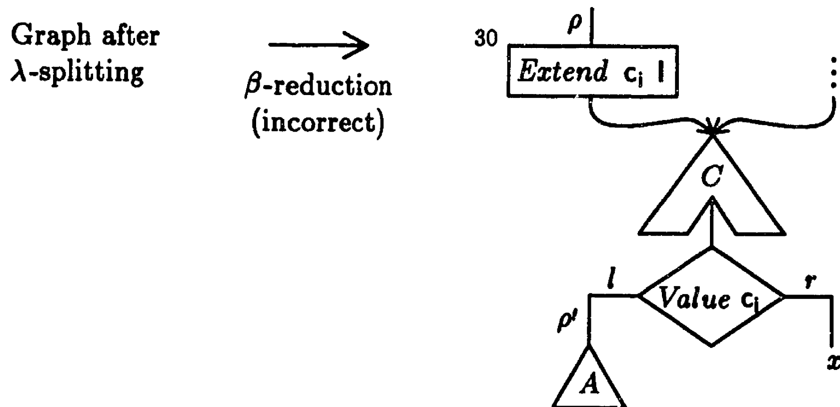


Figure 3.8: Graph after reducing the redex in an obvious way

that β-reduction in our framework should correctly implement β-reduction in the λ-calculus. In other words, suppose we translate the redex in Figure 3.7(b) and its contractum represented by Node 30 in Figure 3.8 relative to the same control environment ρ. Then, the following should hold: If the redex translates to $(\lambda x.M)\,N$, then the contractum translates to $M[x := N]$.

Unfortunately, the answer is no. In general, A will contain some conditional nodes and thus, its translation depends upon the values of certain control variables. To satisfy the correctness criterion, A in the redex as well as in the contractum must be translated using the same values of these control variables. In other words, control variables must follow *static scoping*. It is not obvious that this is the case. A in the redex is translated relative to ρ (see Figure 3.7), whereas A in the contractum is translated relative to ρ' where ρ' is the control environment after passing through the body of the abstraction (see Figure 3.8). In general, ρ and ρ' will be different, since the body of the abstraction may contain box nodes. There is no reason to believe that values of control variables necessary to translate A are the same in ρ and ρ'.

The correctness criterion can be satisfied by translating A in the contractum relative to the control environment ρ. But that implies some way of getting ρ from ρ'. Thus, we introduce a mechanism for saving and restoring control environments.

We extend the representation as follows. First, we introduce a new type of (variable, value) pairs in control environments. Variables this time are $e_1, e_2, \ldots$, which are called *environment variables*. We assume that they are distinct from control variables and λ-calculus variables. Values are control environments themselves. Thus, control environments are no longer flat lists; they are tree-structured. Second we introduce two new functions on control environments—*Save* and *Restore*. Their definitions are as follows:

$$Save\ e_i\ \rho \quad = \quad ((e_i, \rho), \rho)$$

$$Restore\ e_i\ \rho \quad = \quad \rho_1 \text{where } \rho_1 \text{ is the value associated with } e_i \text{ in } \rho$$

Box nodes can also contain these two functions. The translation procedure essentially remains the same; it simply uses the above definitions to translate box node containing the new functions.

The correct way to reduce a β-redex is shown in Figure 3.9. The variable e_i is a fresh environment variable. Further, the boxes containing *Save* and *Restore* are new nodes. The function *Save* before the root of the contractum ensures that the correct control environment for translating A is saved as the value of e_i during the translation. The function *Restore* before A, then, ensures that A is translated relative to the correct control environment.

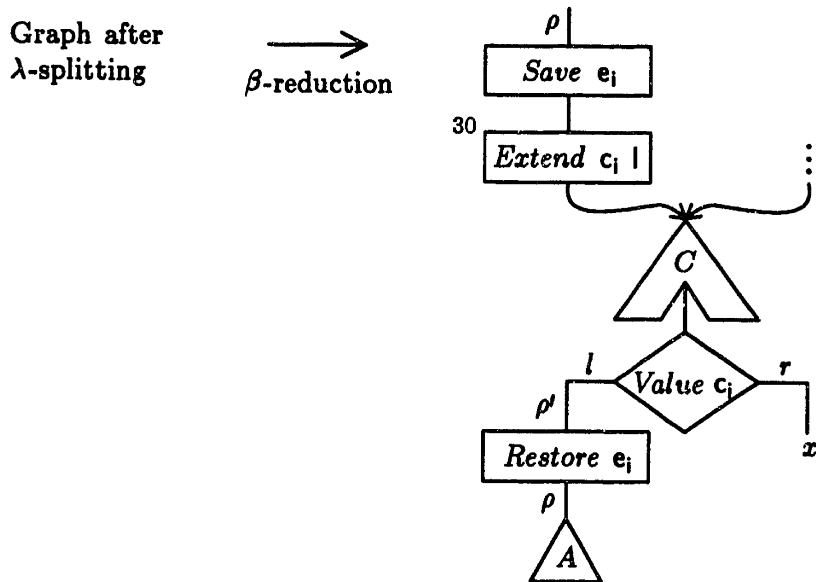


Figure 3.9: Correct way to reduce a β-redex

We must address one more issue, that is, how to avoid capture of free variables. Consider the graph in Figure 3.9. Free variables inside A should not become bound by λ-abstractions inside $C[]$. This is mainly an issue of how one interprets free variables inside A. The relationship of graphs to λ-expressions is described by the translation procedure. Moreover, in graphs, the binding relationship between λ-abstractions and variable occurrences is established essentially by translating the graph. So if the translation procedure interprets free variables inside A in a way so that they don't become bound by abstractions inside $C[]$, then there is no problem. Thus, we define the procedure so that it interprets free variables in A correctly. To do so, we borrow an idea from environment based interpreters—we use control environments to pass correct "binding

information" for free variables of A during the translation. When the translation procedure encounters a λ-abstraction $\lambda x.\cdots$, it puts a pair of the form (x, x') where x' is a completely new variable in the control environment. When it encounters a variable, it simply looks up its new name. The environment saving-restoring mechanism as described earlier, then, ensures that correct binding information for free variables of A is available while translating A. We emphasize that no renaming of bound variables is done in reducing a redex.

3.5 Translation procedure

In this section, we summarize the translation procedure for later reference.

The structure of control environments is given by the following grammar:

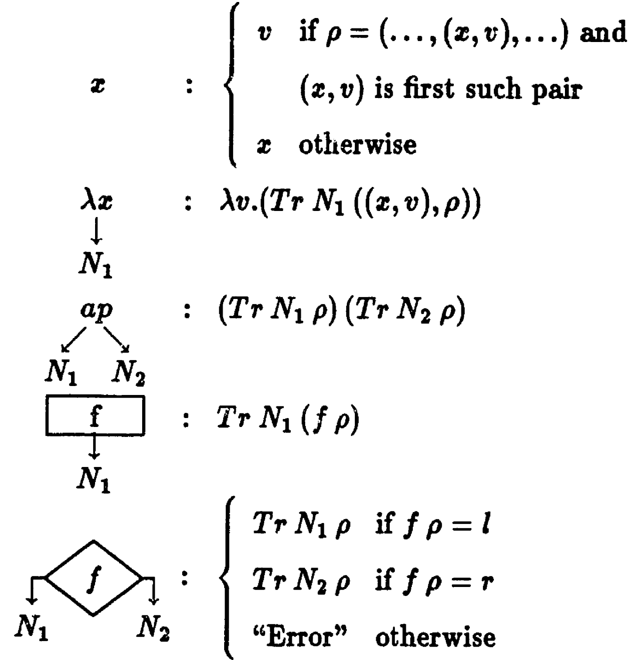
$$\begin{array}{lll} \textit{control-env} & ::= & () \mid (\textit{element}, \textit{control}-\textit{env}) \\ \textit{element} & ::= & (c_i, \textit{val}) \mid (e_i, \textit{control-env}) \mid (x, x') \\ \textit{val} & ::= & l \mid r \end{array}$$

where c_i's are control variables, e_i's are environment variables, and x and v are λ-variables. We will use $\rho, \rho_1, \ldots$ for control environments.

The behavior of functions on control environments introduced so far is summarized below. We will introduce a few more functions in Section 3.7.

$$\begin{array}{lll} \textit{Extend}\ c_i\ \textit{val}\ \rho & = & ((c_i, \textit{val}), \rho) \\ \textit{Value}\ c_i\ \rho & = & \textit{val} \text{ where } \textit{val} \text{ is the value associated with } c_i \text{ in } \rho \\ \textit{Save}\ e_i\ \rho & = & ((e_i, \rho), \rho) \\ \textit{Restore}\ e_i\ \rho & = & \rho_1 \text{where } \rho_1 \text{ is the value associated with } e_i \text{ in } \rho \end{array}$$

The following procedure translates a graph *relative* to a given control environment; the first argument to the procedure is the root of the graph to be translated and the second argument is a control environment.



The expression represented by a graph is the expression obtained by translating the graph relative to the *empty* environment (). Note that the expression represented by a graph that contains no conditional nodes and no box nodes is simply the expression obtained by unraveling the graph (modulo α-renaming).

3.6 Graphs may be cyclic

In this section we show, using an example, that graphs in our representation may be cyclic. However, they always translate into well-formed (hence finite) λ-expressions. The presence of cyclic graphs in our representation simply reflects the fact that sub-expressions to be shared are not necessarily disjoint.

Example 3.2 $\Omega \equiv (\lambda x.x\ x)\ (\lambda x.x\ x)$

To simplify the presentation, we begin with the graph shown in Figure 3.10(a), which is obtained by reducing Ω in an obvious way.

Now consider the reduction of the leftmost redex in this graph using the reduction procedure given in the last section. Figure 3.10(b) shows the graph after the reduction. It obviously contains a cycle, but it translates into Ω (modulo α-renaming). For example, the figure shows a path followed by the translation procedure as well as control environments at various points

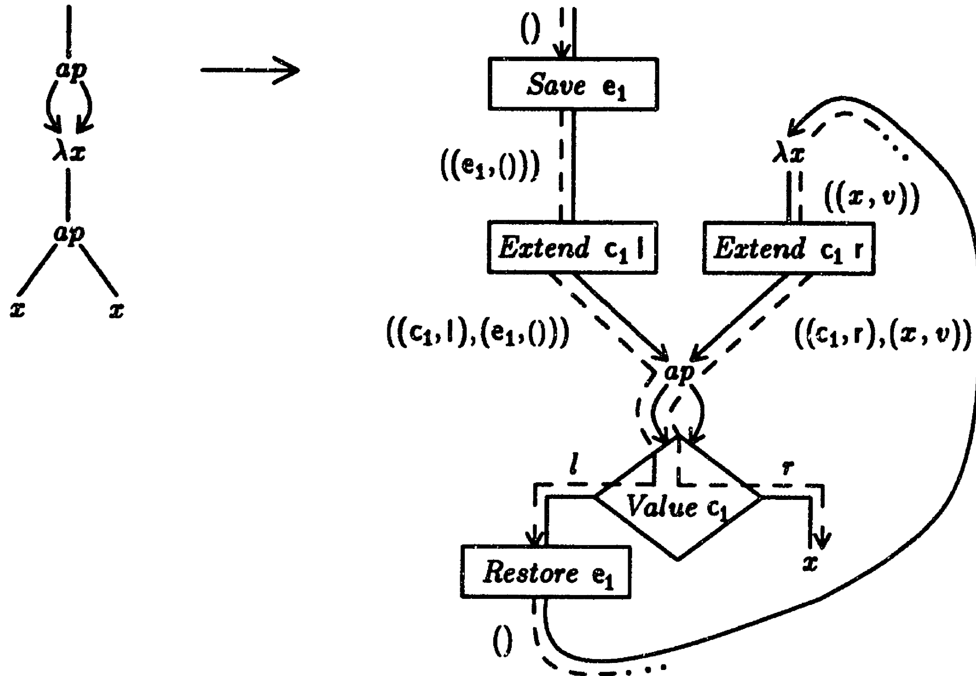


Figure 3.10: Graphs showing reduction of $\Omega \equiv (\lambda x.x\ x)\,(\lambda x.x\ x)$

along the path. The first time the translation procedure reaches the conditional node, the value associated with c_1 is l. So the procedure translates the left hand side of the node. The next time, however, the value associated with c_1 is r. So the procedure translates the right hand side, thus breaking the cycle.

The example also illustrates that the save-restore mechanism introduced in the last section is indeed *necessary*. Consider the next step in the reduction of Ω. Figure 3.11 shows a simplified version of the graph in Figure 3.10(b). If we reduce the redex in Figure 3.11 in the obvious way, *i.e.*, without using the save-restore mechanism, then we get a wrong result. The conditional node in the argument part is translated using different values of c_1 before and after the reduction.

3.7 Converting subgraphs representing β-redexes into β-redexes: New reduction rules

A β-redex in our framework is a subgraph to which the reduction procedure described in Section 3.4 can be applied; see Figure 3.12(a). On the other hand, a subgraph represents a β-redex if it translates (relative to some environment) into a β-redex in the λ-calculus. Because of the presence of conditional and box nodes in graphs, subgraphs that represent β-redexes may not

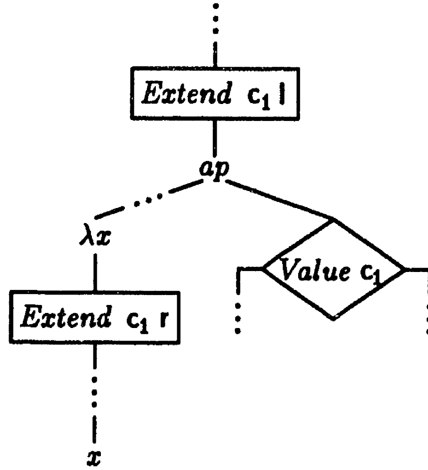


Figure 3.11: Simplified version of the graph in Figure 3.10(b)

be β-redexes; see Figure 3.12(b, c, d). It is clear that the subgraph in part (b) represents a β-redex but is not a β-redex. The subgraph in part (c) also represents a β-redex relative to a control environment in which the value of c_1 is l. The subgraph in part (d) shows the general situation,*i.e.*, an application node and a λ node may be separated by an arbitrary number of boxes and conditional nodes. Given suitable values for various control and environment variables, such a subgraph represents a β-redex. Subgraphs of the form shown in Figure 3.12(d) are first converted into β-redexes by using new reduction rules, which we discuss next.

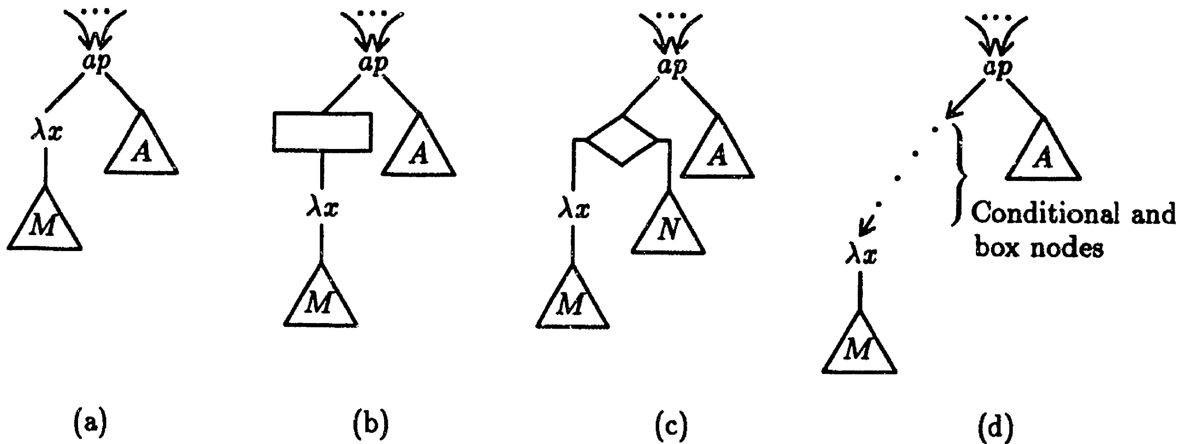


Figure 3.12: (a) A β-redex (b, c, d) Subgraphs that represent β-redexes but are not β-redexes

The new reduction rules try to bring the application node and the λ node parts of a β-redex together by moving box and conditional nodes out of their way. Since the information present in box nodes is used to resolve conditional nodes below them, boxes propagate towards leaves

and conditional nodes propagate towards the root.

The reduction rules should preserve translation and optimality. While presenting the rules, we will informally show that they do. Notice, however, that boxes and conditional nodes have no counterpart in the λ-calculus and thus can be copied freely. On the other hand, if copying of application, λ or variable nodes takes place, then we must show that such copying doesn't destroy optimality.

We introduce three new reduction rules, which are shown in Figure 3.13. All these rules should be read as follows. All occurrences of M (or N) denote the same subgraph, *i.e.*, they are not copies. All other displayed nodes on the right hand side are new nodes. All nodes on the left hand side remain unchanged, though some of them may become inaccessible. All pointers to the root of the redex are redirected to the root of the subgraph on the right hand side. We discuss these rules below.

1. **Pushing a box node past a conditional node:** See Figure 3.13(1). The operation $\circ$ is the usual function composition and behaves as expected, *i.e.*, $(g \circ f)\,\rho = g\,(f\,\rho)$. It is easy to show that the rule preserves translation. Consider the translation of both LHS and RHS relative to the same control environment ρ. LHS translates into

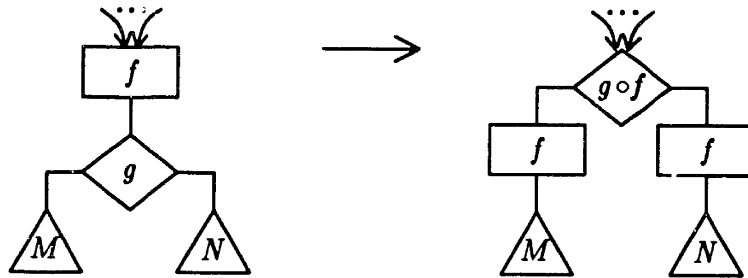
$$\begin{cases} Tr\,M\,(f\,\rho), & \text{if } g\,(f\,\rho) = l \\ Tr\,N\,(f\,\rho), & \text{if } g\,(f\,\rho) = r \end{cases}$$

   The same is, of course, true of RHS. It is also obvious that the rule preserves optimality, since it deals only with box and conditional nodes.

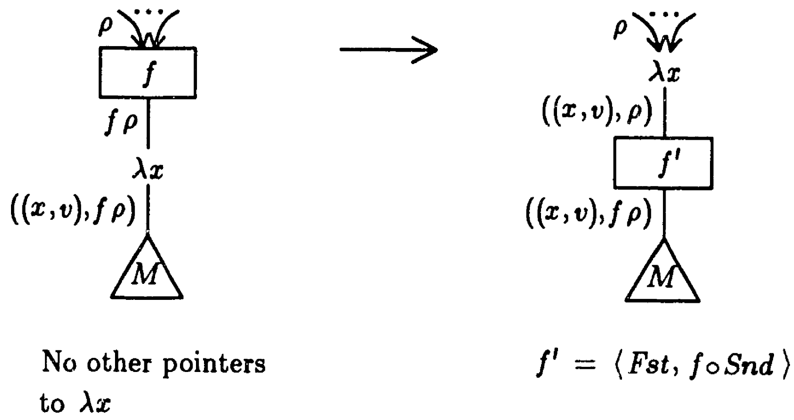
2. **Pushing a box node past a λ node:** See Figure 3.13(2). To see what the function f' should be, we apply the criterion that the rule should preserve translation. Suppose we translate both LHS and RHS relative to ρ. To preserve translation, M on both sides should be translated relative to the same environment. Thus, the function f' should transform $((x, v), \rho)$ into $((x, v), f\,\rho)$. To represent f', we define three new functions: Fst, Snd and $\langle -, - \rangle$. follows:

$$\begin{aligned} Fst\,(element, \rho) &= element \\ Snd\,(element, \rho) &= \rho \\ \langle g, h\rangle\,\rho &= (g\,\rho, h\,\rho) \end{aligned}$$

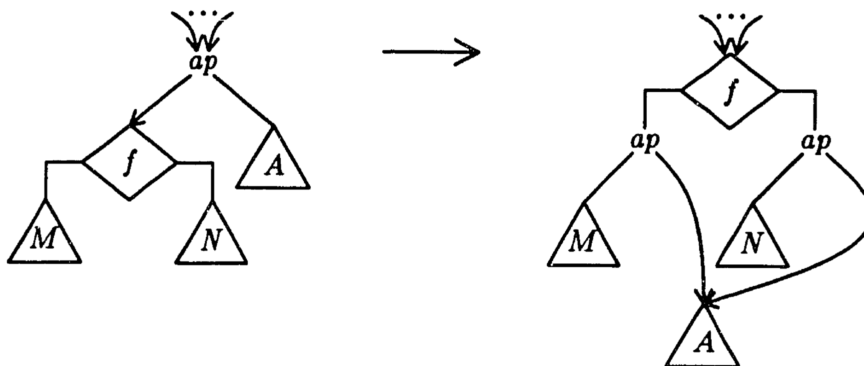
   Then,



(1) Rule to push a box node past a conditional node



(2) Rule to push a box node past a λ node



(3) Rule to push an application node past a conditional node

Figure 3.13: New reduction rules

$$f' = \langle Fst, f \circ Snd \rangle$$

Notice that the rule can be applied only if there are no other pointers to the λ node. In case there are more than one pointers to the λ node, the rule is applied after applying the λ-splitting transformation (see Section 3.4).

The rule obviously preserves optimality, since the rule is applied only when there are no other pointers to λ node. On the other hand, we have to show that condition associated with the rule can be satisfied without destroying optimality, *i.e.*, applying λ-splitting before applying this rule is safe. Actually, applying λ-splitting never destroys optimality because two different pointers to a λ node correspond to two different uses of a λ-abstraction, usually by non-shared redexes.

3. **Pushing an application node past a conditional node:** See Figure 3.13(3). It is obvious that the rule preserves translation. The rule, however, duplicates an application node. So, the question is: Does it preserve optimality? The answer is yes because the copying of application node done by the rule is precisely the copying that is necessary for correctness. These two *ap* nodes can never be reduced together according to Lévy's theory. Consider Example 3.1 introduced at the beginning of this section. Figure 3.14 shows the reduction for the example. In Section 3.2, we argued that copying of Node 50 is necessary. This rule simply does that.

Using these rules, we can convert the subgraph in Figure 3.12(d) into a β-redex as follows. We use the first two rules to move all box nodes between the application node and the λ node past the λ node. Then, we use the last rule to move the application node past all conditional nodes.

3.8 Computation rule

We now know the representation and the procedure to reduce β-redexes. To complete the interpreter, we need to specify the computation rule. We adapt the leftmost computation rule for the λ-calculus to our interpreter. At each step, the interpreter finds the subgraph that represents the leftmost redex in the expression represented by the graph. If necessary, it applies the three rules of the last section to convert the subgraph into a β-redex. Then, it reduces the redex using the procedure described in Section 3.4.

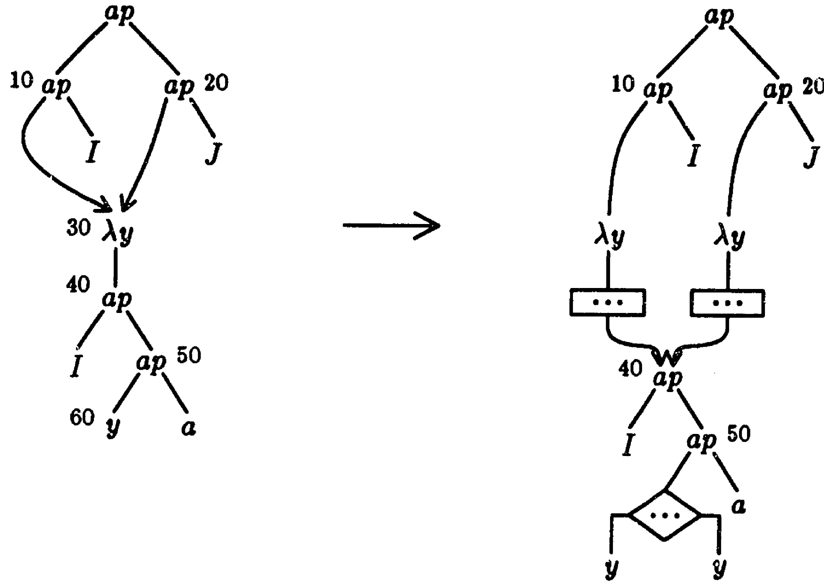


Figure 3.14: Example to illustrate that copying performed in pushing an application node past a conditional node doesn't destroy optimality

The procedure to find the subgraph that represents the leftmost redex in the expression represented by the graph is similar to the translation procedure.

Overall, the interpreter operates as follows. Given a λ-expression, it begins with the tree representation of the expression. It reduces the expression as described above. If the reduction terminates, it translates the final graph into a λ-expression using the translation procedure.

3.9 Deficiencies and improvements

In this section, we presented a preliminary version of the interpreter and argued informally that it is both correct and optimal. This version of the interpreter, however, is not quite satisfactory for the reasons discussed below.

1. *Computation rule*: The procedure to find the "leftmost β-redex", which is similar to the translation procedure, is quite complex and inefficient. We would like an interpreter that applies a set of reduction rules using a simple computation rule, *e.g.*, leftmost.

2. *Translating results into λ-expressions*: Again, the translation is quite complex. We would like the translation of results into λ-expressions to be as simple as unraveling a graph.

3. *Conditional nodes and size of graphs*: A conditional node represents a choice between two subgraphs, the choice being determined by the path followed to reach the node. If there is *only* one path to a conditional node, then one of its two sides is inaccessible. However, such garbage subgraphs, which may be arbitrarily large, remain in graphs because conditional nodes, once introduced, are never eliminated even though some of these nodes don't serve any purpose.

4. *λ-splitting and β-reduction*: Both these rules are not constant time operations. They involve finding all occurrences of the bound variable of a λ-abstraction. The procedure to do so is, of course, similar to the translation procedure. We would like all the rules including these two rules to be constant time operations.

5. *Control environments*: Control environments contain three different types of variable-value pairs. Further, they tend to get big because of the number of control variables introduced during reduction. Successive application of λ-splitting to a λ-abstraction with n pointers introduces $n - 1$ new control variables, all of which are present in a control environment at the same time. Environment variables and λ-variables don't cause problem because only a few of them are present in a control environment at the same time.

   One may argue that the structure and size of control environments are not too important as control environments are used only by the translation procedure. This is true. But, the structure and size of control environments affect functions on environments present in graphs and become important if we want to introduce reduction rules to simplify functions (see below).

   Reduction rules for functions don't work out nicely if control environments are represented as lists of variable-value pairs.

The first three points are related. We introduce several new reduction rules including rules to simplify functions so that conditional nodes can be eliminated as soon as possible. The interpreter applies all the rules in leftmost order. We ensure that by the time the interpreter reaches a conditional node, all the information necessary to resolve the conditional node is present in the condition part of the node. The interpreter simplifies the conditional part to decide whether the node should be replaced by the left or the right side.

To make λ-splitting and β-reduction constant time operations, we represent the association between a λ-abstraction and occurrences of its bound variables *explicitly* using pointers. Thus,

the interpreter never has to search for occurrences of the bound variable of a λ-abstraction.

To address the last point, we change the structure of control environments in two ways. First, we encode more succinctly the information present in a control environment. The idea is that the information present in a control environment is, in some sense, related to λ-abstractions, and all the information related to a single λ-abstraction in the initial expression can be represented as a single structure. The structure of control environments becomes similar to the structure of environments in environment-based interpreters. Second, to make reduction rules for functions simple, we eliminate variable names by switching to DeBruijn's notation for λ-expressions.

3.10 New structure of control environments

In the last section, we pointed out that the structure of control environments is not very satisfactory for several reasons including the following. First, control environments contain three different type of variable-value pairs. Second, they tend to get big because successive application of λ-splitting to a λ-abstraction with n pointers introduces $n-1$ new control variables, all of which are present in a control environment at the same time. In this section, we present a new structure for control environments that is more succinct and doesn't have the problems mentioned above.

The idea is this. The information present in a control environment is, in some sense, related to λ-abstractions. Control variables are introduced when applying λ-splitting to λ-abstractions. Environment variables are introduced when using λ-abstraction in β-reduction. λ-variables are, of course, related to λ-abstractions. Moreover, λ-abstractions don't get created; a λ-abstraction in the expression at any step of the reduction comes from a λ-abstraction in the initial expression. Thus, the information present in a control environment can be related to λ-abstractions in the initial expressions. All the information related to one λ-abstraction in the initial expression can be kept as a single structure as the value of one control variable. Furthermore, names of bound variables can themselves be used as the names of control variables.

We describe the new structure of control environments in three steps. Changing the structure of control environments, of course, changes the way λ-splitting and β-reduction is done. So we also discuss how these operation are done using the new structure of control environments.

Combining control variables introduced for a λ-abstraction

Consider a λ-abstraction with more than two pointers, say three. Then, successive application of λ-splitting introduces two new control variables; see Figure 3.15. Consider all the control

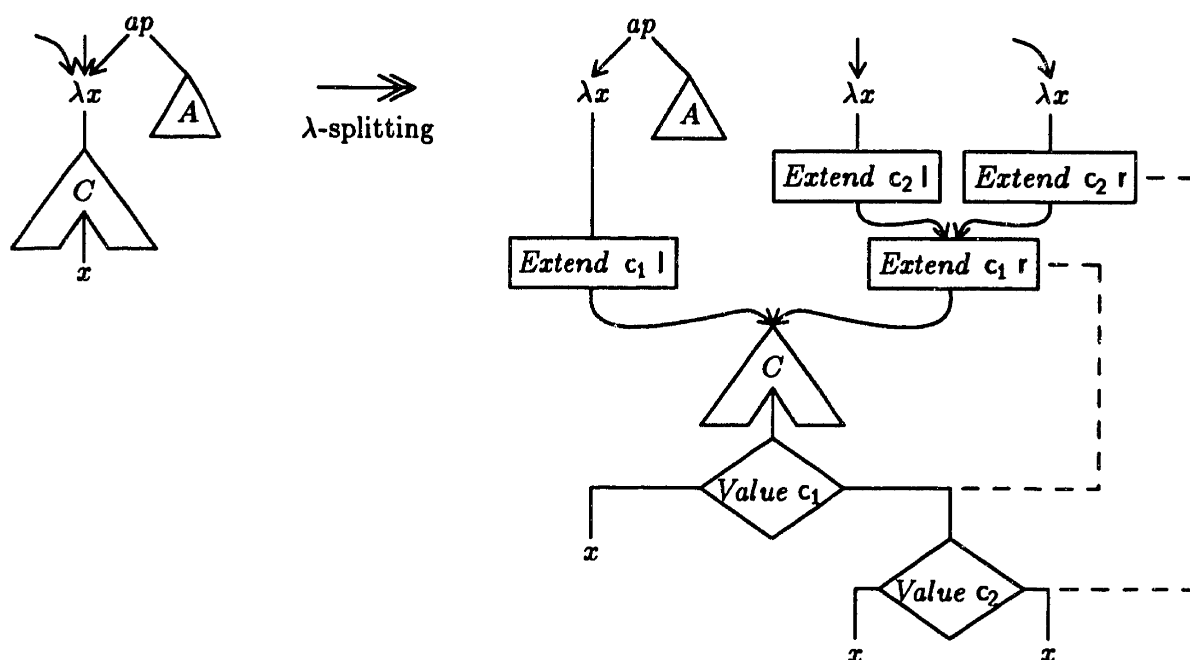


Figure 3.15: Successive application of λ-splitting introduces several new control variables

variables introduced in λ-splitting a λ-abstraction in the initial expression. The important point to note about this group of control variables is that there is a complete nesting of def-use of these variables. The variable introduced last on a path is used by the first conditional encountered, second last by the second conditional encountered and so on; see Figure 3.15. Thus, we don't need "random access" to control variables in a group; a stack-based approach suffices.

Thus, we represent the values of control variables introduced for a λ-abstraction simply as a sequence of l and r, which we call a *path*. A path behaves like a stack, and we use equivalents of push, pop, and top to deal with paths.

We associate a control variable with each λ-abstraction in the initial expression. Control environments now contain pairs of the form $(c_x, path)$ where c_x is a control variable associated with a λ-abstraction.

λ-splitting is done as shown in Figure 3.16 The function *Push* is like adding a new pair to a control environment except that it adds a new value to the path component of an already existing pair.

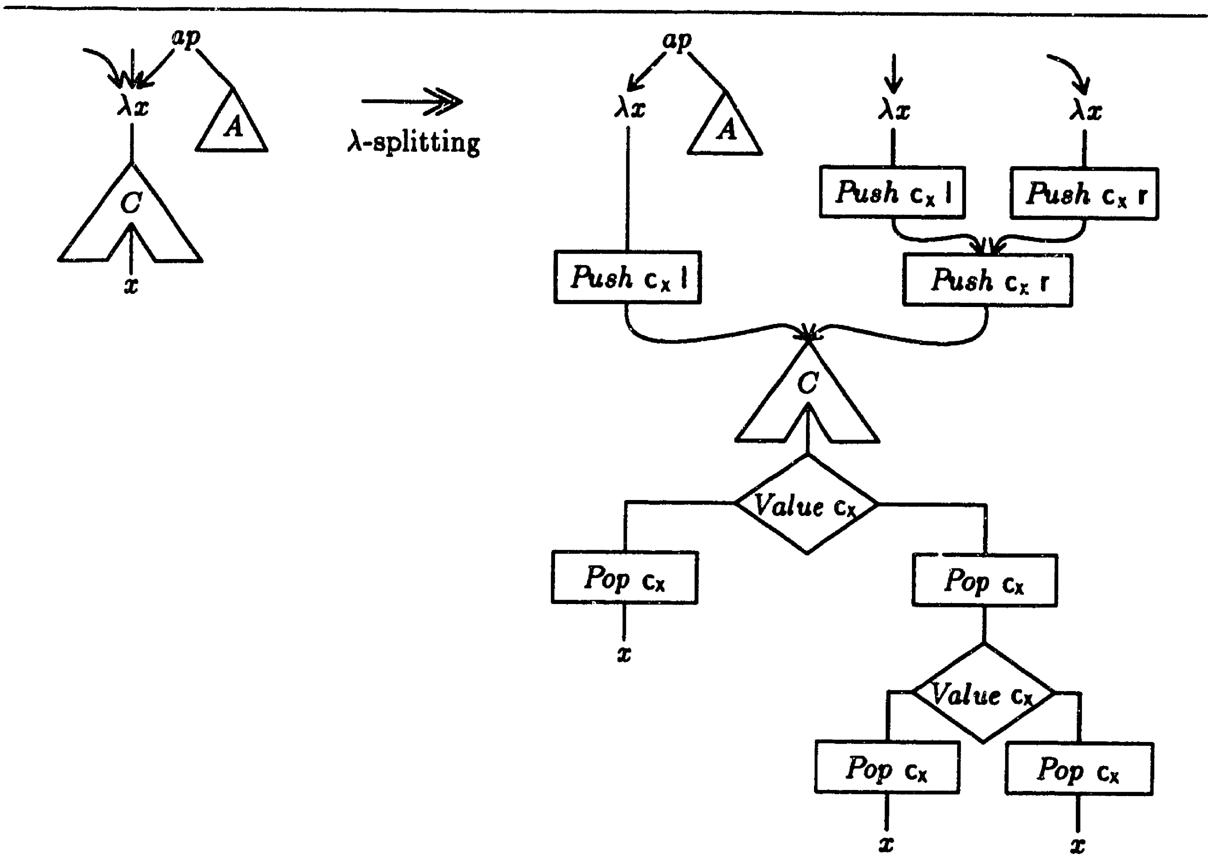


Figure 3.16: λ-splitting with new structure of control environments

$$Push\, c_x\, v(...(c_x, path)...) \;=\; (...(c_x, v\, path)...)$$

where v is either l or r. To simplify things later, the function at a conditional node returns a path and not the first symbol in the path. We define the translation procedure so that it bases its decision on the first symbol of the path returned by applying the function to a control environment. The definition of *Pop* is as follows:

$$Pop\, c_x\, (...(c_x, v\, path)...) \;=\; (...(c_x, path)...)$$

It is used to ensure that the correct value for resolving a conditional is always available as the first symbol of the path.

The initial value of a control variable is the *empty* path ϵ. It is established by the translation procedure when translating a λ-abstraction, *i.e.*,

$$Tr\; \lambda x.M\; \rho \;=\; \lambda x.(Tr\; M\; ((c_x, \epsilon), (x, x'), \rho))$$

where c_x is the control variable associated with the λ-abstraction.

Combining control and environment variables

Consider the reduction of a *beta*-redex in Figure 3.15. Since the reduction eliminates the abstraction part of the redex, we must explicitly establish the initial value of the control variable associated with the λ-abstraction by adding a pair (c_x, ϵ) to control environments. The reduction of a redex also introduces an environment variable for saving and restoring environments. There is, however, no need for keeping control and environment variables separate; we can combine the two by putting more structure on the value of a control variable.

We make the value of a control variable to be of the from $[path, env]$. The β-reduction is now done as shown in Figure 3.17. The definitions of *Save'* and *Restore'* are as follows:

$$Save'\, c_x\, \rho \;=\; ((c_x, [\epsilon, \rho]), \rho)$$

$$Restore'\, c_x\, \rho \;=\; \rho' \qquad \text{where } \rho' \text{ is the environment part of the value associated with } c_x \text{ in } \rho$$

Using λ-variables as control variables

We make the control variable associated with a λ-abstraction to be the same as the bound variable of the abstraction. Further, we modify the translation as follows:

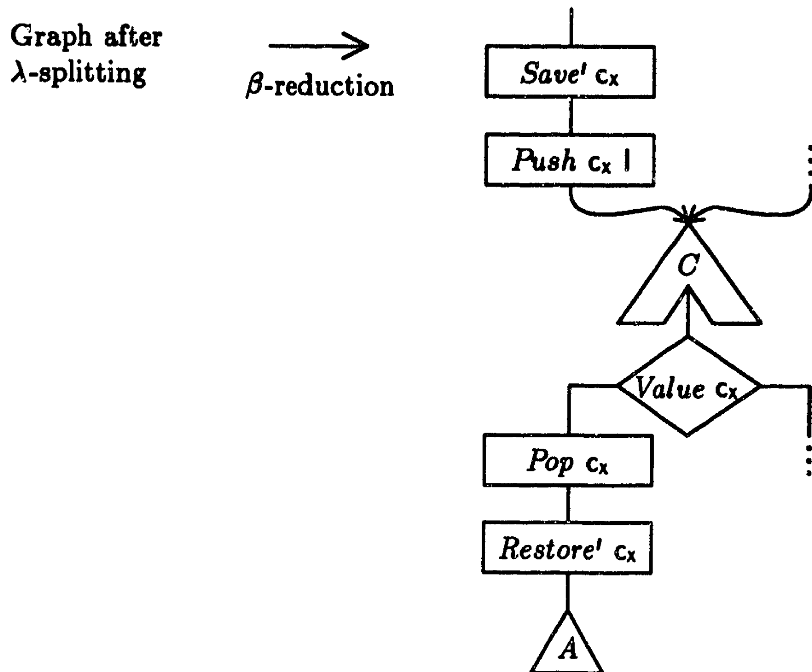


Figure 3.17: β-reduction: combining control and environment variables

$$Tr\ \lambda x.M\ \rho = \lambda v.(TrM\ ((x, [\epsilon, x']), \rho))$$

$$Tr\ x\ \rho = x', \text{if } [\epsilon, x] \text{ is the value associated with } x \text{ in } \rho.$$

Summarizing, the new structure of control environments is as follows:

control-env	::=	() | (*element*, *control-env*)
element	::=	(λ-*variable*, *value*)
value	::=	[*path*, *control-env*] | [*path*, λ-*variable*]
path	::=	ϵ | *l path* | *r path*

This structure of control environments is similar to the structure of environments in environment-based interpreters. The rough analogy is that a value of the form [*path*, *control−env*] corresponds to a closure, and the *path* component of the value corresponds to an explicit encoding of the pointer to the expression part of the closure.

The new structure of control environments fixes some of the deficiencies discussed earlier. Moreover, it is easy to get a positional notation for accessing elements of control environments, which makes it easier to specify reduction rules for functions on control environments.

3.11 Summary

In this chapter, we introduced the idea of sharing contexts. We developed an interpreter for the λ-calculus that is capable of sharing contexts. We discussed the graph representation and the reduction rules used by the interpreter. At the end of the chapter, we discussed some of its deficiencies and suggested a number of improvements.

Chapter 4

Optimal Interpreter using De Bruijn's Notation

In this chapter, we switch to De Bruijn's notation for λ-expressions in order to incorporate some of the improvements suggested in the last chapter. In Section 4.1, we describe De Bruijn's notation. In Section 4.2, we discuss λ-splitting and β-reduction using De Bruijn's notation. To describe our interpreter, we need a way of describing graphs, graph reduction rules, and how reduction rules are applied. We use an extended version of the approach proposed by Barendregt *et. al.* [7], and we describe their approach in Section 4.3. Finally, in Section 4.4, we give a complete description of the interpreter.

4.1 De Bruijn's Notation for the λ-calculus

De Bruijn's notation [12] is a name-free notation for the λ-calculus. In De Bruijn's notation, a variable occurrence in a λ-expression is replaced by a natural number that denotes the number of λ-abstractions between the variable occurrence and its binding λ-abstraction. The assignment of numbers to *free* occurrences of variables in an expression is done by simply assuming that the free variables of the expressions are implicitly bound, in some order, by λ-abstractions at the beginning of the expression.

Example 4.1 $M \equiv (\lambda x.(\lambda z.z\, x)\,(x\, z))\,((\lambda t.t)\, y)$

The structure of M becomes more apparent if we consider M as a tree, see Figure 4.1(a). In Figure 4.1(b), the implicit bindings are shown by heavy dots, and the number associated with a variable occurrence is written below the occurrence. Notice that all occurrences of a variable bound by a λ-abstraction need not have the same number; for example, see the numbers

assigned to the two occurrences of x. The expression in De Bruijn's notation corresponding to M is as follows:

$$M_{DB} \equiv (\lambda.(\lambda.0\ 1)\ (0\ 2))\ ((\lambda.0)\ 0)$$

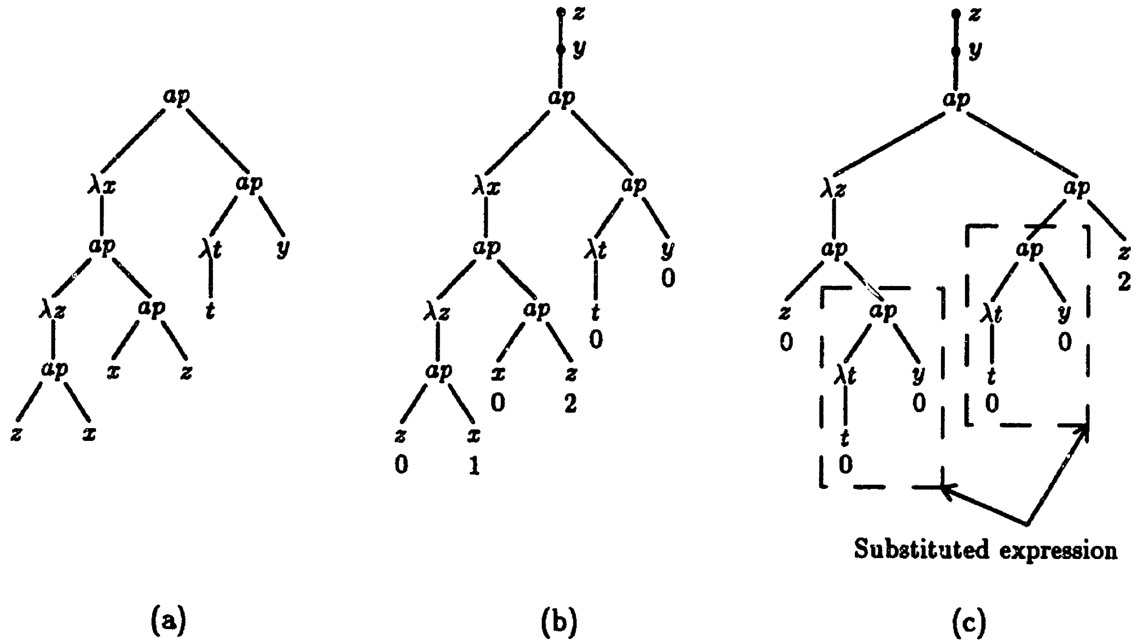


Figure 4.1: (a) Tree representation of M (b) Implicit bindings and reference depths (c) Term after naively reducing the leftmost redex in (b)

As in the λ-calculus, a redex $(\lambda.M)\ N$ is reduced by substituting N for each occurrence of a number that is "bound" by the λ-abstraction. Although there are no variable names, the substitution operation is still not naive. Suppose we reduce the leftmost redex in M_{DB} using naive substitution. Figure 4.1(c) shows the term after the reduction; we have kept names for convenience. Keeping the interpretation of numbers in mind, the leftmost occurrence of y has now become bound by λz, and the rightmost occurrence of z is no longer bound by any λ-abstraction. Thus, some adjustment of numbers is necessary to properly model the β-reduction in the λ-calculus. Essentially, the reference depths of certain occurrences must be recomputed. This recomputation is part of the substitution operation, and exactly how it is done will be described later.

We now give the formal description of Be Bruijn's notation for the λ-calculus.

Definition 4.2 The set Λ_{DB} of λ-terms in De Bruijn's notation is given by the following grammar:

$$
\begin{aligned}
Term \;::=\;& n \in \mathbf{N} \text{ where } \mathbf{N} \text{ is the set of natural numbers} \\
& \mid Term\,Term \mid \lambda.Term
\end{aligned}
$$

Usual conventions about parentheses apply.

Definition 4.3 Let M be a λ-term in DeBruijn's notation. An occurrence of a number n in M is *free* in M iff the number of λ-abstractions on the path from the root of M to the occurrence of n is no greater than n.

An occurrence of a number n in M that is not free in M is said to be *bound* in M. Moreover, it is said to be bound by the $(n+1)^{th}$ λ-abstraction above it.

Note that it is not necessary to define free and bound occurrences, all that is needed is an explicit definition of the substitution operation.

We now formalize translations suggested at the beginning of the section between classical λ-calculus (with only β-rule) and De Bruijn's notation for the λ-calculus. Translations are defined relative to an ordered list of variables, *i.e.*, a formal environment.

Definition 4.4

1. For any $M \in \Lambda$ such that $FV(M) \subset \{x_0, \ldots, x_n\}$, its translation into De Bruijn's notation, denoted by $M_{DB,(x_0,\ldots,x_n)}$, is defined as follows:

$$
\begin{aligned}
x_{DB,(x_0,\ldots,x_n)} &= i, \quad \text{where } i \text{ is the minimum number such that } x = x_i \\
(M\,N)_{DB,(x_0,\ldots,x_n)} &= M_{DB,(x_0,\ldots,x_n)}\, N_{DB,(x_0,\ldots,x_n)} \\
(\lambda x.M)_{DB,(x_0,\ldots,x_n)} &= \lambda.M_{DB,(x,x_0,\ldots,x_n)}
\end{aligned}
$$

2. The λ-term corresponding to a term M in De Bruijn's notation relative to a formal environment $(x_0, \ldots, x_n)$ is denoted by $M_{\lambda,(x_0,\ldots,x_n)}$ and is defined as follows:

$$
\begin{aligned}
k_{\lambda,(x_0,\ldots,x_n)} &= \begin{cases} x_k & \text{if } k \le n \\ \text{“Error”} & \text{otherwise} \end{cases} \\
(M\,N)_{\lambda,(x_0,\ldots,x_n)} &= M_{\lambda,(x_0,\ldots,x_n)}\, N_{\lambda,(x_0,\ldots,x_n)} \\
(\lambda.M)_{\lambda,(x_0,\ldots,x_n)} &= \lambda x.M_{\lambda,(x,x_0,\ldots,x_n)} \\
& \quad \text{where } x \text{ is a new variable different from } x_0, \ldots, x_n
\end{aligned}
$$

We now give the definition of substitution operation in De Bruijn's formalism.

Definition 4.5 For terms $M, N \in \Lambda_{DB}$ and a natural number m, the operation $M[m := N]$, called *substitution*, is defined as follows:

$$
\begin{aligned}
n[m := N] &= \begin{cases} n & \text{if } n < m \\ U_0^n(N) & \text{if } n = m \\ n - 1 & \text{if } n > m \end{cases} \\
(M_1\, M_2)[m := N] &= M_1[m := N]\, M_2[m := N] \\
(\lambda.M)[m := N] &= \lambda.(M[m+1 := N])
\end{aligned}
$$

where the operation $U_m^n(N)$, called *increment by n*, is defined as follows:

$$
\begin{aligned}
U_m^n(k) &= \begin{cases} k & \text{if } k < m \\ k + n & \text{if } k \geq m \end{cases} \\
U_m^n(N_1\, N_2) &= U_m^n(N_1)\, U_m^n(N_2) \\
U_m^n(\lambda.N) &= \lambda.(U_{m+1}^n(N))
\end{aligned}
$$

Definition 4.6 β-rule in De Bruijn's notation is as follows:

$$(\beta): \qquad (\lambda.M)\, N \longrightarrow M[0 := N]$$

A few comments about the above two definitions. The parameter m in the recursive definitions of substitution and increment operations keeps track of the number of λ-abstractions encountered so far. It is used to decide if an occurrence of a number is free or bound. Consider, for example, the definition of substitution operation. The three cases in the first clause correspond to the occurrence of n being bound in M, the occurrence being free in M but bound in $\lambda.M$, and the occurrence being free in $\lambda.M$.

The substitution operation replaces an occurrence of n that is free in $\lambda.M$ by $n-1$ to reflect the fact that one λ-abstraction (*i.e.*, the one involved in the reduction) between the occurrence and its binding λ-abstraction disappears after the reduction. The operation $U_0^n(N)$ adds n to each occurrence that is free in N. The reason is that, after the reduction, n new λ-abstractions appear between a free occurrence in N and its binding λ-abstraction.

The reader may verify his understanding of the substitution and lifting operations by reducing the expression M_{DB} given earlier. (It contains all the interesting cases.)

The following proposition describes the precise correspondence between λ-calculus and De Bruijn's notation:

Proposition 4.7 *Let $M, N \in \Lambda$, and suppose $FV(M) \subset \{x_0, \ldots, x_n\}$. Let $A, B \in \Lambda_{DB}$, and let ρ be a formal environment such that $A_{\lambda,\rho}$ doesn't lead to error. Then,*

1. $M \xrightarrow[\beta]{} N \Rightarrow M_{DB,(x_0,\ldots,x_n)} \xrightarrow[\beta]{} N_{DB,(x_0,\ldots,x_n)}$

2. $A \xrightarrow[\beta]{} B \Rightarrow A_{\lambda,\rho} \xrightarrow[\beta]{} B_{\lambda,\rho}$

3. $(M_{DB,(x_0,\ldots,x_n)})_{\lambda,(x_0,\ldots,x_n)} =_\alpha M$

4. $(A_{\lambda,\rho})_{DB,\rho} \equiv A$

Moreover, the reduction on the two sides of (1) or (2) are, in some sense, of the same redex (modulo translation).

We end this section with some remarks. De Bruijn's notation can be viewed as an implementation of lexical scoping using dynamic scoping. Thinking in terms of environments, the De Bruijn number of a variable occurrence specifies the position in an environment at which the value associated with the occurrence is to be found. Environment-based interpreters implement lexical scoping using *closures* to preserve the environment in which an expression should be evaluated. In De Bruijn's notation, a term can always be evaluated in the environment at the time of evaluation (dynamic scoping) because terms themselves are modified by changing numbers to implement lexical scoping.

Sharing of subexpressions in direct implementations of De Bruijn's notation is impossible because of the way β-reduction changes certain numbers inside substituted copies of the argument part and the body of abstraction part of the redex. Sharing of the argument part is impossible because numbers inside different copies of the argument part are, in general, changed differently. Furthermore, if the abstraction part is shared, it becomes necessary to copy the whole of the abstraction; sharing of free expression like Wadsworth's interpreter is impossible because numbers changed inside the body are precisely the ones that denote free occurrences. In [13], De Bruijn showed that changing of numbers inside substituted copies of the argument part of a redex can be done lazily.

4.2 λ-splitting and β-reduction in De Bruijn's notation

In Section 3.10, we described λ-splitting and β-reduction using the new structure of control environments. In this section, we rephrase that discussion in terms of De Bruijn's notation.

As mentioned at the end of the last section, the De Bruijn number of a variable occurrence specifies the position in an environment at which the value associated with the occurrence is to be found. So control environments are now represented simply as lists of values. The following grammar defines the structure of control environments:

$$
\begin{array}{lcl}
\mathit{control\text{-}env} & ::= & () \mid (\mathit{control\text{-}env}, \mathit{value}) \\
\mathit{value} & ::= & [\mathit{path}, \mathit{control\text{-}env}] \mid [\mathit{path}, n] \\
\mathit{path} & ::= & \epsilon \mid l\,\mathit{path} \mid r\,\mathit{path}
\end{array}
$$

where () denotes the empty list.

Values of the form $[path, n]$ correspond to values of the form $[path, v]$, which were used to interpret free variables correctly. Here they are used to simulate changing of numbers that is part of the β-reduction in De Bruijn's notation. A value of the form $[path, n']$ in position n says that the number n should be changed to n'.

Control environments are structured so that the value associated with the last λ-abstraction encountered is in the last position. In other words, the structure of control environments reflects the nesting structure of λ-abstractions from bottom to top; see Figure 4.2. Thus, number 0

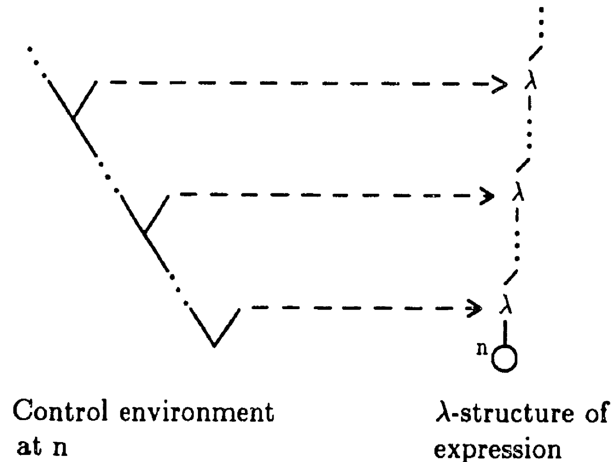


Figure 4.2: Relationship of the structure of control environments with the nesting structure of λ-abstraction

refers to the last value, number 1 to the last but one value, and so on. Notice that this structure is the reverse of what we have been working with so far.

We also make some notational changes. Functions on control environments, which were written inside box and conditional nodes, will now be displayed explicitly as another subgraph. Box nodes and conditional nodes will carry labels ε and *cond*, respectively. See Figure 4.3.

Figure 4.4 shows schematically λ-splitting and β-reduction in De Bruijn's notation. Subscripts on functions indicate the relationship with functions introduced in Section 3.10. Below we describe how various functions in the figure are represented.

- f_{pushl} and f_{pushr}: These functions put l or r at the beginning of the *path* component of

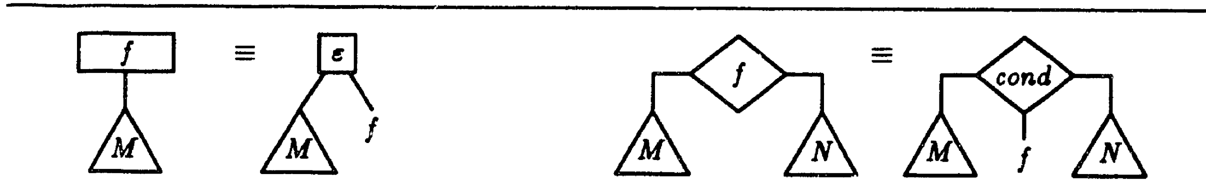


Figure 4.3: Notational changes

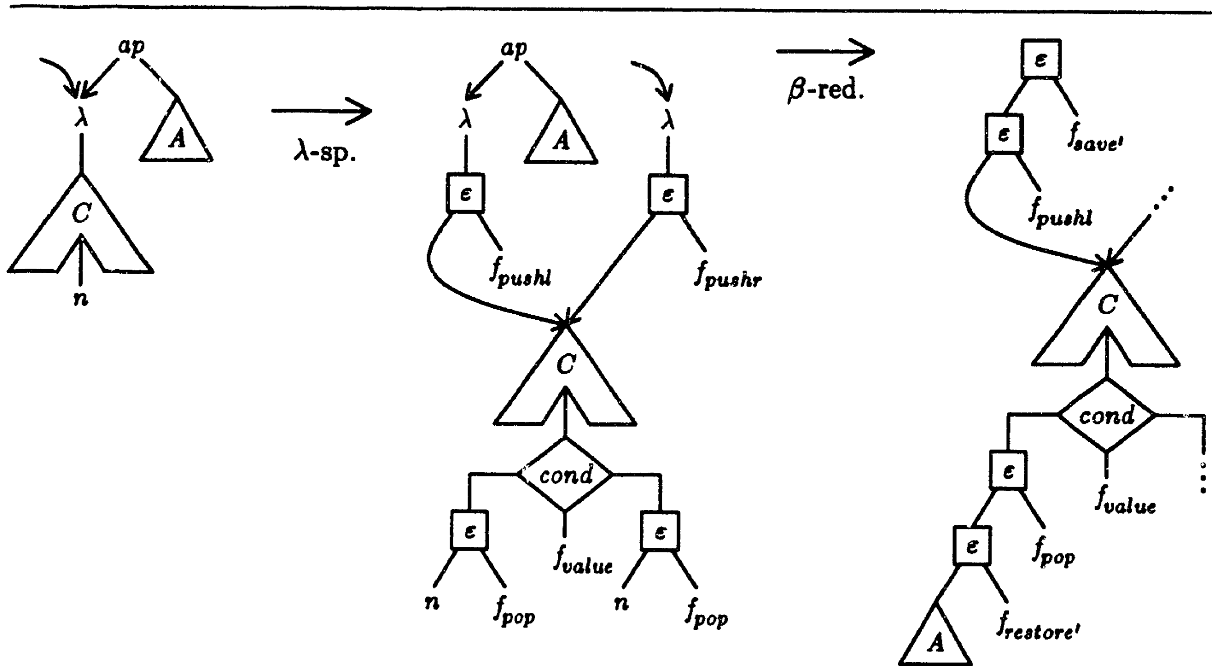


Figure 4.4: λ-splitting and β-reduction using DeBruijn's notation

the value associated with the λ-abstraction, which will always be in the last position in control environments. So we represent them as follows:

$$\begin{aligned} f_{pushl} &\equiv \langle Fst, L \circ Snd \rangle \\ f_{pushr} &\equiv \langle Fst, R \circ Snd \rangle \end{aligned}$$

Functions Fst, Snd and $\langle -, - \rangle$ are essentially the same as in Section 3.7 except that their types have changed because of the reverse structure of control environments.

$$\begin{aligned} Fst\,(\rho, value) &= \rho \\ Snd\,(\rho, value) &= value \\ \langle f, g \rangle\, \rho &= (f\,\rho, g\,\rho) \end{aligned}$$

where ρ is a control environment. The definitions of L and R are as follows:

$$\begin{aligned} L\,[path, x] &= [l\,path, x] \\ R\,[path, x] &= [r\,path, x] \end{aligned}$$

where x is either a number or a control environment.

- $f_{save'}$: The function adds a new value of the form $[\epsilon, \rho]$ at the end of a control environment ρ. Thus, we represent it as

$$\begin{aligned} f_{save'} &\equiv \langle Id, Empty \rangle, \text{ where} \\ Id\,\rho &= \rho, \text{ and} \\ Empty\,\rho &= [\epsilon, \rho] \end{aligned}$$

All the remaining functions have a feature in common—they all access the nth value from control environments. Moreover, the discussion in Section 3.10 implies that function f_{pop} accesses the nth value, changes it, and then puts the environment back again. Immediately after that, either the translation of n or the function $f_{restore'}$ accesses the nth value and discards the rest of the environment. In other words, these functions perform a lot of duplicate work.

To avoid the duplicate work, we encode De Bruijn numbers in a special way. A DeBruijn number can be viewed in two different ways. Viewed in terms of the substitution operation, it denotes the "place" where substitution takes place. Viewed in terms of environments, it specifies the position in an environment at which the value associated with the occurrence is to be found (as mentioned earlier). Since we use both views in our interpreter, we encode

n as $\varepsilon(i, Snd \circ Fst^n)$ where Fst^n denotes n-fold composition of Fst. The symbol i is not a variable; it is the only symbol of its kind and is used to denote places where substitution takes place. The function $Snd \circ Fst^n$, which we abbreviate as $n!$, is used to access control environments. It is also used to establish binding relationship between i's and λ-abstractions using control environments. This encoding of n is based on a similar encoding used by Curien in his Categorical Combinator Logic (see [11, 10]).

With this encoding of numbers, we represent all places bound by a λ-abstraction as a single i node. Furthermore, we make the binding relationship explicit by adding a *binding pointer* from the λ node to the i node. In this way, we avoid the time consuming search for all occurrences bound by a λ-abstraction. λ-splitting and β-reduction become constant time operations.

Figure 4.5 shows λ-splitting and β-reduction in the new notation. All the remaining functions operate on n^{th} value and are represented as follows:

- f'_{value}: The function does nothing. We represent it as *identity* function on values. Rather than introducing a new function, we make function Id introduced earlier polymorphic so that it works on both values and control environments.

- f'_{pop}: We represent it as function Pop defined below:

$$Pop\,[v\,path, x] \;=\; [path, x]$$

  where v is either l or r.

- $f'_{restore'}$: It is represented as function Env defined below:

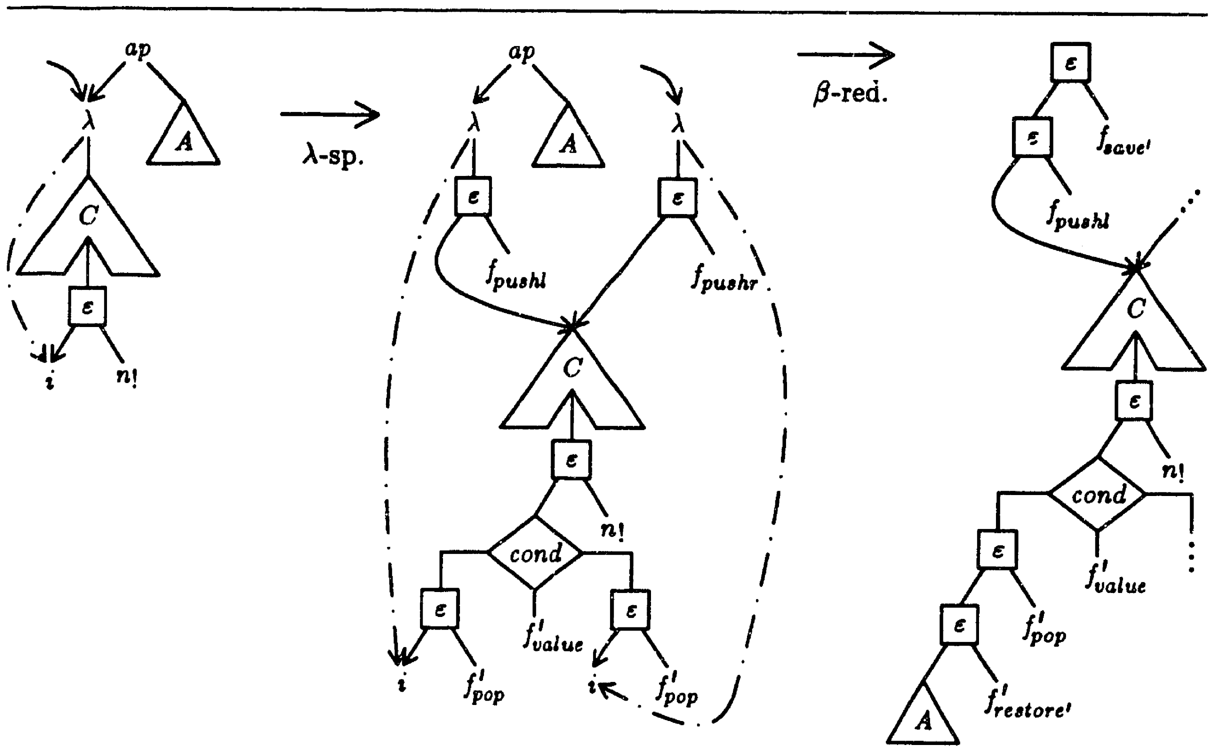
$$Env\,[path, \rho] \;=\; \rho$$

  It is an error to apply Env to a value whose second component is not an environment.

Notice that we no longer have control environments at every point in translating a graph; at some points we have control environments, at others we only have values.

4.3 A notation for graph reduction

To describe our interpreter, we need a way of describing graphs, graph reduction rules, and how reduction rules are applied. Barendregt *et. al.* [7] gave an approach for describing certain



— · — · → Binding pointer

$n! \equiv Snd \circ Fst^n$

Figure 4.5: λ-splitting and β-reduction using new encoding of n and binding pointers

types of graph reduction systems (see also Staples's work [31, 32, 33]). Although their approach doesn't provide the most general description of graph rewriting, it is quite operational compared to some of the other approaches, especially the algebraic approach of Ehrig and others [14, 28]. Furthermore, the development of their approach parallels the way term rewriting systems (TRSs) are described. We will follow, with a few extensions, their approach to describe our interpreter.

In this section, we describe the approach. We assume some familiarity with TRSs and will use the following TRS as an example.

Example 4.8 The signature contains two function symbols: *ap* of arity 2 and S combinator of arity 0. The only reduction rule is the well known rule for S combinator.

$$ap(ap(ap(S, x), y), z) \longrightarrow ap(ap(x, z), ap(y, z))$$

Like (closed) terms over a signature, we define *term graphs* over a signature. A term graph is a rooted, directed, ordered graph in which each node is labelled by a function symbol. Further, if a node is labelled with a function symbol of arity k, then it has k sons, called its *successors.* Acyclic graphs correspond to terms in an obvious way. We will permit cyclic graphs, though they don't correspond to anything in the term world.

Definition 4.9 Let Σ be a signature. A *labelled graph* over Σ is a triple $(N, lab, succ)$ involving

- a set N of *nodes*,
- a labelling function $lab : N \rightarrow \Sigma$, and
- a successor function $succ : N \rightarrow \text{List}(N)$ such that for all $n \in N$, if n is labelled with a function symbol of arity k, then $succ(n) = (n_1, \ldots, n_k)$.

A *term graph* over Σ is a rooted labelled graph. A term graph is written as a quadruple $(N, lab, succ, r)$ where r is the *root* of the graph.

For a term graph g, the components are often denoted by $N_g, lab_g, succ_g$, and r_g. The ith component of $succ(n)$ is denoted by $succ(n)_i$.

Notice that we don't require that every node in a term graph is reachable from the root. When we draw pictures of term graphs, the topmost node is usually the root, and $succ(n)$ for node n are ordered from left to right.

The usual graph theoretic notions of paths, cycles, etc. apply to labelled graphs. Further, we have the usual notion of subgraphs:

Definition 4.10 Let $g = (N, lab, succ)$ be a labelled graph and let n be a node in g. The *subgraph of g rooted at n* is the term graph $(N', lab', succ', n)$ where

- $N' = \{n' \in N \mid \text{there is path from } n \text{ to } n'\}$, and
- lab' and $succ'$ are restrictions of lab and $succ$ to N'.

We denote this graph by $g|n$.

To define graph reduction rules, we need a notion analogous to open terms, *i.e.*, terms with variables. Instead of using variables, it is more convenient to use the notion of *empty nodes* (see Staples [31]).

Definition 4.11 An *open labelled graph* is a triple $(N, lab, succ)$ like a labelled graph, except that lab and $succ$ are only required to be partial functions on N. A node on which lab and $succ$ are undefined is called an *empty* node . We use the symbol ○ to denote empty nodes. An open term graph is defined similarly.

Definition 4.12 A *graph reduction rule* is a triple (g, n, n'), where g is an open labelled graph and n and n' are nodes of g. The graph $g|n$ is the left hand side and the graph $g|n'$ is the right hand side of the rule.

There is an easy way to convert certain types of term rewrite rules to graph rewrite rules. Let $t_l \longrightarrow t_r$ be a left-linear term rewrite rule (*i.e.*, no variable in t_l occur more than once). The corresponding graph rewrite rule (g, n, n') is obtained as follows. The graph g is the union of the tree representations of t_l and t_r, sharing those empty nodes that represent the same variable in t_l and t_r. Nodes n and n' are simply the roots of trees representing t_l and t_r, respectively. Consider, for example, the term rule in Example 4.8. The corresponding graph rewrite rule is shown in Figure 4.6. The arrow from n to n' is not part of the graph; it is included to visually separate the left hand side from the right hand side of the rule.

We now describe how to apply graph reduction rules. In TRSs, the application of a rule $t_l \longrightarrow t_r$ to a term t involves two steps. The first step is to identify a subexpression of t that matches the left hand side of the rule, *i.e.*, a subexpression of the form $\sigma(t_l)$ where σ is a substitution for variables. Terms of the form $\sigma(t_l)$ are called redexes. The second step is to replace the redex by the term $\sigma(t_r)$. The corresponding notions for graphs are a little more involved. To define the notion of a redex in a graph, we need the notion of homomorphisms between graphs.

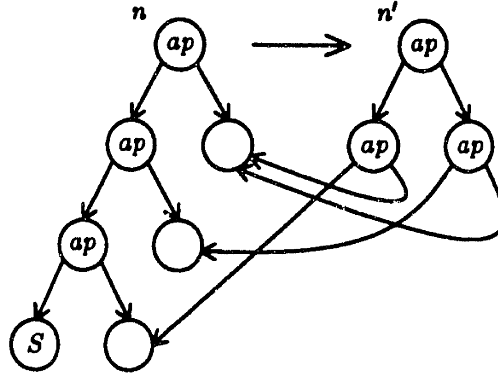


Figure 4.6: The graph rewrite rule corresponding to the term rule for S combinator

Definition 4.13 Given two graphs $g_1 = (N_1, lab_1, succ_1, r_1)$ and $g_2 = (N_2, lab_2, succ_2, r_2)$, a *homomorphism* from g_1 to g_2 is a map $f : N_1 \rightarrow N_2$ that preserves labels, successors and their order. That is,

$$\begin{aligned} lab_2(f(n)) &= lab_1(n) \\ succ_2(f(n)) &= \mathbf{f}(succ_1(n)) \quad \text{where } \mathbf{f}(n_1, \ldots, n_k) = (f(n_1), \ldots, f(n_k)). \end{aligned}$$

A homomorphism from an open graph to a graph is defined as above except that the structure preserving conditions are required to hold only at *nonempty* nodes. In other words, empty nodes can map to any node (*cf.* substitutions in TRSs).

Figure 4.7(a) and (b) show situations in which homomorphism exist between graphs. The arrow shows the mapping for the root node of the graph; mapping for the root node completely determines mapping for other nodes. Figure 4.7(c) shows a situation in which there is no homomorphism between graphs. Notice that a homomorphism from g_1 to g_2 preserves the sharing in g_1.

Definition 4.14 A *redex* in a graph g_0 is a pair (r, f) where r is a graph rewrite rule (g, n, n') and f is a homomorphism from $g|n$ to g_0. The homomorphism f is called an *occurrence* of r.

Once a redex in a graph has been identified, it is reduced in three phases. We use the following redex as an example: (g, n, n') is simply the graph rule for S combinator, g_0 and the operation of f on n are shown in Figure 4.8. The figure also shows graphs after each phase of the reduction.

1. In the first phase, called *build* phase, an isomorphic copy of parts of $g|n'$ that are not

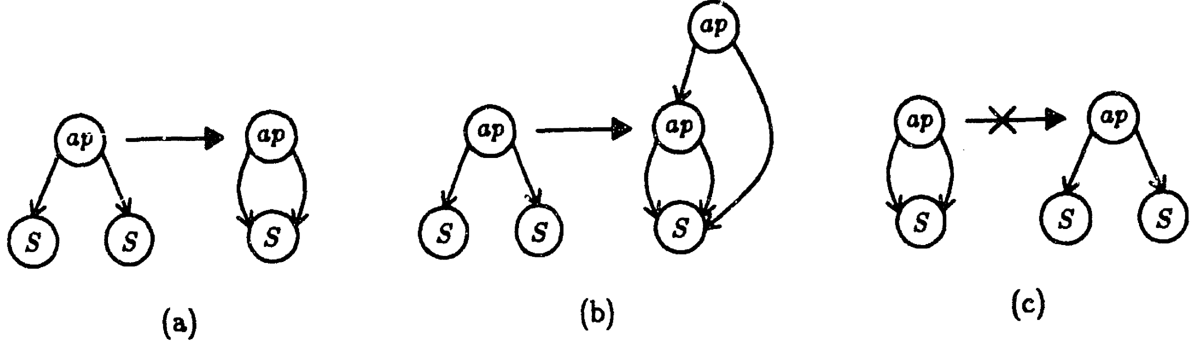


Figure 4.7: (a) and (b) Situations in which homomorphisms between graphs exist. (c) A situation in which there in no homomorphism between graphs.

contained in $g|n$ is added to g_0 with *lab*, *succ*, and root defined in the natural way. We call this graph g_1.

2. In the second phase, called *redirection* phase, all edges of g_1 pointing to $f(n)$ are redirected to the copy of n', giving a graph g_2. If the root of g_1 is $f(n)$, then the root of g_2 is n'. Otherwise, the root of g_2 is the root of g_1.

3. In the last phase, called *garbage collection* phase, all nodes not accessible from the root of g_2 are removed, giving a graph g_3, which is the result of the reduction.

   Notice that an *ap* node and two S nodes that were part of the redex graph remain after garbage collection as they are still accessible from the root. The other two *ap* nodes of the redex graph has been garbage collected.

Definition 4.15 Let $((g,n,n'),f)$ be redex in graph g_0. Graphs g_1 (after the build phase), g_2 (after the redirection phase) and g_3 (after the garbage collection phase) are defined as follows. We use the abbreviation N_{new} for $N_{g|n'} - N_{g|n}$; nodes in N_{new} are the new nodes added to g_0.

1. $g_1 = (N_1, lab_1, succ_1, r_1)$ where

$$
\begin{aligned}
N_1 &= \text{the disjoint union of } N_{g_0} \text{ and } N_{new} \\
lab_1(m) &= \begin{cases} lab_{g_0}(m) & \text{if } m \in N_{g_0} \\ lab_g(m) & \text{if } m \in N_{new} \end{cases} \\
succ_1(m)_i &= \begin{cases} succ_{g_0}(m)_i & \text{if } m \in N_{g_0} \\ succ_g(m)_i & \text{if } m, succ_g(m)_i \in N_{new} \\ f(succ_g(m)_i) & \text{if } m \in N_{new} \text{ and } succ_g(m)_i \in N_{g|n} \end{cases} \\
r_1 &= r_{g_0}
\end{aligned}
$$

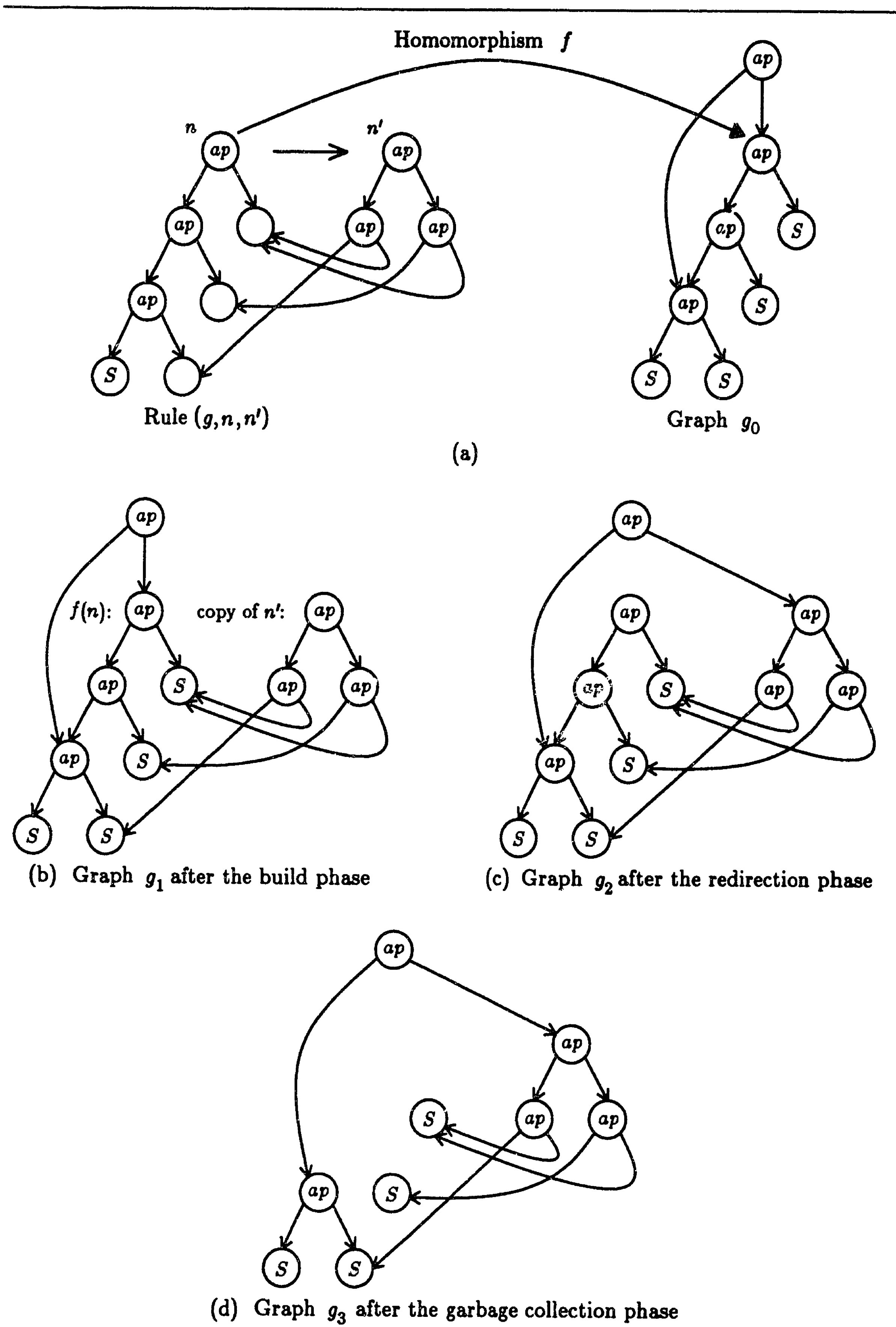


Figure 4.8: Applying a graph reduction rule

We write $g_1 = g_0 +_f (g, n, n')$.

2. The next step is to replace in g_1 all references to $f(n)$ by references to n'. We define a general substitution operation for any term graph $h = (N, lab, succ, r)$ and any two nodes a and b of h. Graph $h[a := b]$ is the term graph $(N', lab', succ', r')$ where

$$\begin{aligned}
N' &= N \\
lab'(n) &= lab(n) \\
succ'(n)_i &= \begin{cases} b & \text{if } succ(n)_i = a \\ succ(n)_i & \text{otherwise} \end{cases} \\
r' &= \begin{cases} b & \text{if } r = a \\ r & \text{otherwise} \end{cases}
\end{aligned}$$

Using this definition, $g_2 = g_1[f(n) := n']$.

3. Finally $g_3 = g_2|r_2$. For any term graph h, we write $GC(h) = h|r_h$.

Collecting all these definitions, the result of reducing a redex $((g, n, n'), f)$ in a graph g_0 is the graph

$$GC((g_0 +_f (g, n, n'))[f(n) := n']).$$

A graph reduction system simply consists of a set of graphs over a signature and a set of graph reduction rules. Normal forms, one-step reduction $\longrightarrow$, its reflexive transitive closure $\twoheadrightarrow$, *etc.* are defined in the same way as for TRSs.

4.4 The interpreter

In this section, we give a complete and concise description of the interpreter. We present the interpreter as a graph reduction system consisting of a set of graphs and a set of reduction rules to transform graphs. To complete the description of the interpreter, we give a normalizing computation rule for applying the reduction rules. Furthermore, to relate our interpreter to the λ-calculus, we give procedures to translate λ-expressions in De Bruijn's notation into graphs and *vice versa*.

The graph reduction system underlying the interpreter will be described in two parts. The first part describes functions on environments and reduction rules for these functions. The second part describes the rest.

4.4.1 Functions on control environments

Before we go to graphs, we describe functions and reductions rules for these functions using the usual term notation.

Definition 4.16 *Functions on control environments* are expressions built using the following signature, called Σ_f.

- $Id, Fst, Snd, Empty, Env, L, R$ and Pop of arity 0
- $\circ$ and $\langle -, - \rangle$ of arity 2

In other words,

$$f \quad ::= \quad Id \mid Fst \mid Snd \mid Empty \mid L \mid R \mid Pop \mid Env \mid f \circ f \mid \langle f, f \rangle$$

$\mathbf{F}_c$ will denote the set of functions on environments. Variables $f, g, h, \ldots$ range over $\mathbf{F}_c$.

Definition 4.17 F_c is the following set of reduction rules. The rules follow easily from the definitions of these functions given in Section 4.2.

(Ass): $(f \circ g) \circ h \longrightarrow f \circ (g \circ h)$

(IdL): $Id \circ f \longrightarrow f$

(IdR): $f \circ Id \longrightarrow f$

(Fst): $Fst \circ \langle f, g \rangle \longrightarrow f$

(Snd): $Snd \circ \langle f, g \rangle \longrightarrow g$

(Dpair): $\langle f, g \rangle \circ h \longrightarrow \langle f \circ h, g \circ h \rangle$

(PopL): $Pop \circ L \longrightarrow Id$

(PopR): $Pop \circ R \longrightarrow Id$

(Env): $Env \circ Empty \longrightarrow Id$

To avoid applying the associativity rule in the reverse direction, it is necessary to have the following variations of the last three rules.

(PopL'): $Pop \circ (L \circ f) \longrightarrow f$

(PopR'): $Pop \circ (R \circ f) \longrightarrow f$

(Env'): $Env \circ (Empty \circ f) \longrightarrow f$

In fact, the rules $(PopL)$ and $(PopR)$ are never used by the interpreter in reducing λ-expressions to normal forms, as L or R always occur in composition with other functions. At this point, however, we impose no conditions on L and R.

The first five rules are identical to the similarly named rules used in Curien's CCL [11].

We now convert the reduction system described above to a graph reduction system using the approach described in the last section.

Definition 4.18 An F_c graph is an *acyclic* term graph over Σ_f.

Definition 4.19 F_g is the set of reduction rules shown in Figures 4.9 and 4.10. All these rules have the form (g, n, n') where g is the whole graph, and n and n' are as indicated. As usual the arrow from n to n' is not part of g.

Since F_c graphs are acyclic, they correspond to functions written in the term notation in an obvious way—the term represented by an F_c graph is obtained by simply unraveling the graph.

4.4.2 λ_{fc} graphs

We first give the signature over which graphs are constructed.

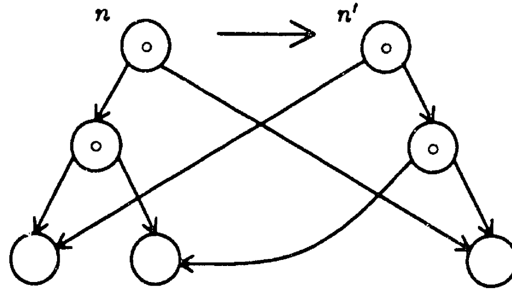
Definition 4.20 The signature Σ_{fc} is the union of Σ_f and the following signature, called Σ_c:

- i of arity 0
- λ, *ap* and ε of arity 2
- *cond* of arity 3

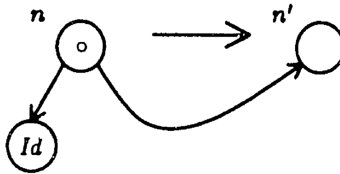
Note that the function symbol λ has arity 2 to accommodate binding pointers, see the discussion in Section 4.2. Thus, a λ node will have two successors; the first successor represents the body of the λ-abstraction, and the second successor represents the i node bound by the λ-abstraction.

The graphs that are of interest to us are actually graphs over Σ_c in which F graphs occur as parts of ε and *cond* nodes. They must satisfy two additional requirements, both of which have to do with the binding relationship between λ nodes and i nodes. First, the binding pointer from a λ node must always point to an i node, since it doesn't make much sense for a λ-abstraction to bind any other type of expression. Second, two λ nodes should not have binding pointers to the same i node. That is, occurrences bound by two λ-abstractions should

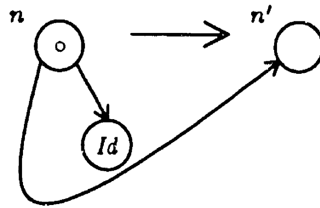
(Ass):



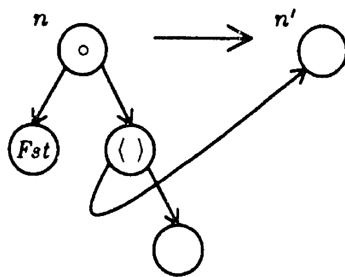
(IdL):



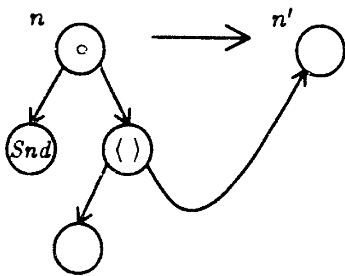
(IdR):



(Fst):



(Snd):



(Dpair):

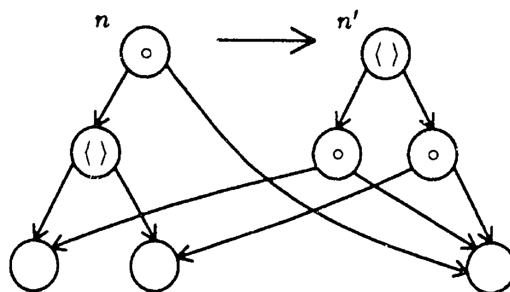
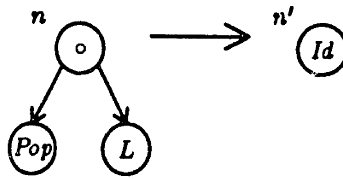
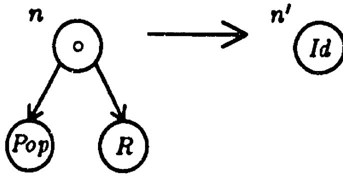


Figure 4.9: F rules

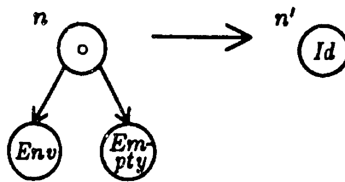
(PopL):



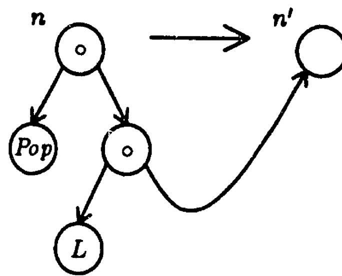
(PopR):



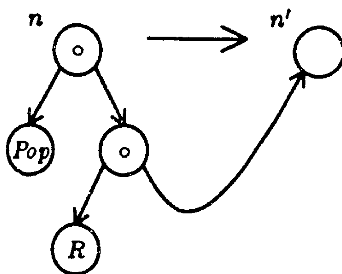
(Env):



(PopL'):



(PopR'):



(Env'):

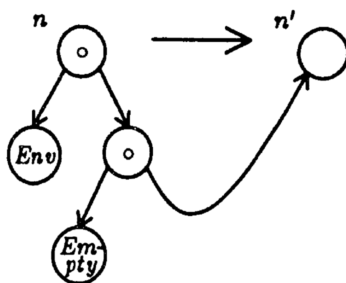


Figure 4.10: F rules (cont.)

be shared only if the λ-abstractions themselves are shared. The last requirement ensures that λ-splitting and β-reduction don't have unintended effect. Hence the following definition:

Definition 4.21 A λ_{fc} graph $G = (N, lab, succ, r)$ is a term graph over Σ_{fc} such that the following conditions are satisfied.

1. $lab(r) \in \Sigma_c$.

2. For any node $n \in N$,

   (a) if $lab(n) = \lambda$, then $lab(succ(n)_1) \in \Sigma_c$ and $lab(succ(n)_2) = i$,

   (b) if $lab(n) = ap$, then $lab(succ(n)_1)$ and $lab(succ(n)_2)$ are in Σ_c,

   (c) if $lab(n) = \varepsilon$, then $lab(succ(n)_1) \in \Sigma_c$ and $succ(n)_2$ is the root of an F_c graph,

   (d) if $lab(n) = cond$, then $succ(n)_1$ is the root of an F_c graph, and both $lab(succ(n)_2)$ and $lab(succ(n)_3)$ are in Σ_c.

   Figure 4.11 shows these conditions. The figure also shows the conventions used in drawing λ_{fc} graphs, most of which we have already been using. Note that the successors of a node are ordered from left to right except for *cond* nodes. For a *cond* node, the order of successors from left to right is 2, 1, 3, which is the convention we have been following.

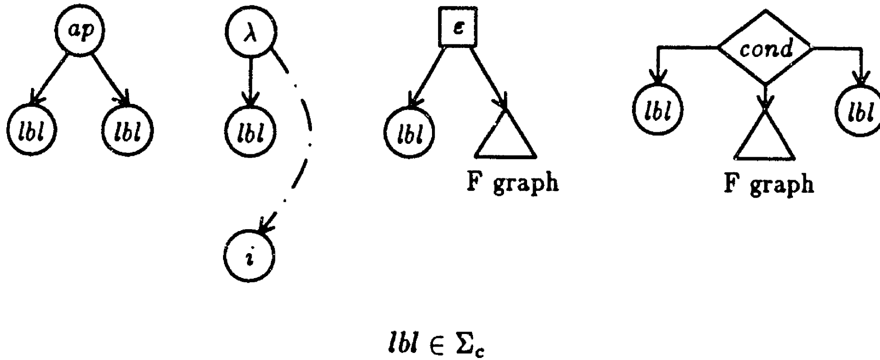


Figure 4.11: Conditions satisfied by λ_{fc} graphs

3. For any two λ labelled nodes n and n', $succ(n)_2 \neq succ(n')_2$.

Note that there is no requirement that λ_{fc} graphs be acyclic.

4.4.3 Translations

In this section, we give procedures to translate λ-expressions in De Bruijn's notation into graphs and *vice versa.* There is no unique way to represent λ-expressions in De Bruijn's notation as graphs. The representation we give here is the simplest one; it corresponds, in some sense, to the tree representation of terms.

Definition 4.22 The *initial graph representation* of a λ-expression in De Bruijn's notation is obtained as follows:

1. Construct the tree representation of the expression.
2. Construct a λ_{fc} graph that is identical to the tree except that a number k is replaced by the tree representation of $\varepsilon(i, Snd \circ Fst^k)$. Furthermore, keep the second successor of λ nodes undefined.

   If a number was bound by a λ-abstraction in the tree, then we shall consider the occurrence of i introduced for the number to be bound by the corresponding λ node in the λ_{fc} graph.
3. Merge all i nodes bound by a λ node into one node, and make this node to be the second successor of the λ node. If there are no i nodes bound by a λ node, then introduce a dummy i node.

Note that the initial graph corresponding to a DeBruijn term has the following properties:

1. It is acyclic and contains no *cond* nodes.
2. All ε nodes occur in the form $\varepsilon(i, Snd \circ Fst^k)$ for some k.
3. There are no occurrences of F graphs other than the ones indicated in (2).

The procedure to translate graphs into De Bruijn expressions is similar to the one given in Section 3.5 except that the translation is now relative to a function on control environments. The procedure takes the root node of a graph and a function on control environments as inputs and returns the expression represented by the graph (relative to the function). Variable g on the left hand side of a clause stands for an F graph whereas the same variable on the right hand side stands for the term represented by the F graph.

$$
\begin{array}{lll}
Tr\,n\,f & = & \textbf{case } n \textbf{ of} \\
 & i & : \begin{cases} k & \text{if } f \xrightarrow[F]{} k! \\ \text{“Error”} & \text{otherwise} \end{cases} \\
 & \begin{array}{c}\lambda \\ \swarrow\ \searrow \\ n_1 \quad i\end{array} & : \lambda.(Tr\,n_1\,\langle f \circ Fst, Snd\rangle) \\
 & \begin{array}{c}ap \\ \swarrow\ \searrow \\ n_1 \quad n_2\end{array} & : (Tr\,n_1\,f)\,(Tr\,n_2\,f) \\
 & \begin{array}{c}\boxed{\varepsilon} \\ \swarrow\ \searrow \\ n_1 \quad g\end{array} & : Tr\,n_1\,g \circ f \\
 & \begin{array}{c} cond \\ \downarrow \quad \downarrow \quad \downarrow \\ n_1 \quad g \quad n_2\end{array} & : \begin{cases} Tr\,n_1\,f & \text{if } g \circ f \xrightarrow[F]{} L \text{ or } L \circ h \\ Tr\,n_2\,f & \text{if } g \circ f \xrightarrow[F]{} R \text{ or } R \circ h \\ \text{“Error”} & \text{otherwise} \end{cases}
\end{array}
$$

The De Bruijn expression represented by a graph is simply the expression obtained by translating the graph relative to the function Id. Not all λ_{fc} graphs, however, can be translated into expressions in De Bruijn's notation. Because of cyclic graphs, the translation procedure when applied to a graph may not terminate. Moreover, if it terminates, it may return "Error".

Thus, we define a subset of λ_{fc} graphs, called *λ_{fc} graphs that represent De Bruijn expressions.* These are λ_{fc} graphs that satisfy two conditions. First, a graph in this subset can be translated into a De Bruijn expression. The second condition is this. In λ_{fc} graphs, there are two kinds of binding relationship between λ nodes and i nodes. One is the relationship described by binding pointers, which is the one used by the interpreter. The other corresponds to the binding relationship in De Bruijn expressions, and it is expressed in terms of functions. To establish correctness of the interpreter, it is necessary that the two binding relationships agree. See Definition 6.30 in Chapter 6.

Notice that the initial graph representation of a De Bruijn expression is a λ_{fc} graph that represents a De Bruijn expression.

The interpreter, of course, never uses the above translation procedure, not even to translate results into λ-expressions. If we start with the initial graph corresponding to a DeBruijn expression and reduce it using the strategy described in Section 4.4.5, then the final graph is very much like the initial graph representation of an expression in that it satisfies the three conditions given earlier. This graph can be translated into an expression in De Bruijn's notation very easily. We simply unravel the graph, ignore binding pointers from λ nodes, and replace

each occurrence of $\varepsilon(i, Snd \circ Fst^k)$ by the the k.

4.4.4 Reduction rules

Section 4.4.1 gives reduction rules for functions on control environments. In this section, we give the remaining reduction rules.

To simplify drawings of reduction rules, we display F graphs using the term notation; such a term actually stands for the tree representation of the term. Variables $f, g, \ldots$ in a rule stand for empty nodes that, in applying the rule, must map to nodes labelled with function symbols in Σ_f. Multiple occurrences of the same variable stand for the same node. An occurrence of such a variable in a term stands for a pointer to the node represented by the variable.

Rules for λ-splitting and β-reduction

These rules have been discussed extensively. Figure 4.12 shows the two rules, now named (λ-sp) and (β_{fc}). They use an extended version of the notation introduced in the last section. Below we describe what these rules mean and how they are applied to a graph.

We write the rule (λ-sp) in the following form:

$$(\lambda\text{-sp}): \quad (g, n, n', m, n'', n_1, n_1')$$

where g is the whole graph except the arrow between λ nodes, and nodes n, m, *etc.* are as shown in the figure.

To identify an occurrence of this rule in a graph g_0, we construct a mapping $f : N_{g|n} \cup m \rightarrow N_{g_0}$ such that

- the mapping f is a homomorphism from $g|n$ to g_0, and
- it maps node m to a node that has a pointer to node $f(n)$.

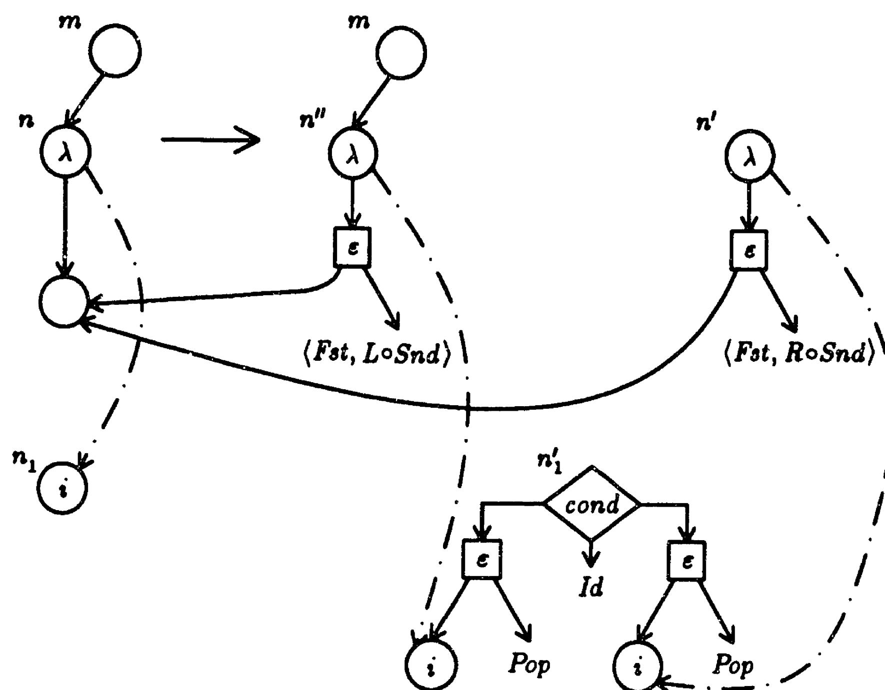
The node $f(m)$ can have any label and $f(n)$ can be any successor of $f(m)$.

The reduction, as usual, proceeds in three phases. For the build phase the set of new nodes is given by

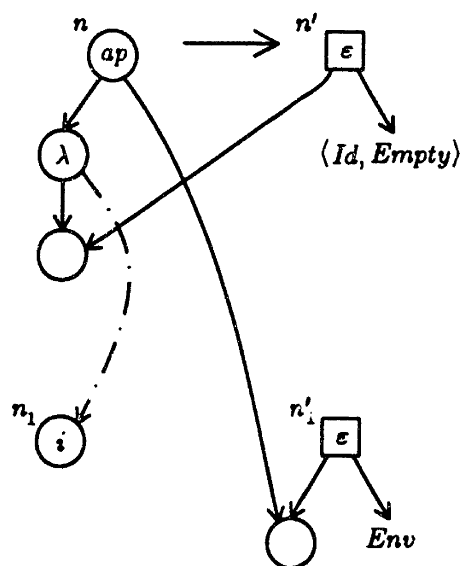
$$N_{new} = (N_{g|n'} \cup N_{g|n''}) - N_{g|n}.$$

Notice that the node m, though displayed twice in the figure, is not in the set of new nodes. Redirection of pointers is done in the following way.

(λ-sp):



(β_{fc}):



No other pointers to λ-node

Figure 4.12: Rules (λ-sp) and (β_{fc})

1. The pointer from $f(m)$ to $f(n)$ becomes the pointer from $f(m)$ to the copy of n''; all other pointers to $f(n)$ are redirected to the copy of n'.

2. All pointers to $f(n_1)$ are redirected to the copy of n_1'.

Although the last redirection implies that the binding pointer of $f(n)$ points to a *cond* node, this is not a problem because $f(n)$ becomes a garbage node. The garbage collection phase remains the same.

The result of the reduction will be denoted by

$$GC((g_0 +_f \lambda\text{-sp})[(f(m), f(n)) := (f(m), n''), f(n) := n'][f(n_1) := n_1'])$$

where (a, b) denotes a pointer from a to b.

The rule (β_{fc}) is a little simpler. We write it as

(β_{fc}). $\quad (g, n, n', n_1, n_1')$

Notice that the rule has a condition attached to it—it can be applied only if there are no other pointers to the λ node. The rule is applied as if it were of the form (g, n, n') with two exceptions. Suppose the homomorphism f defines an occurrence of this rule in a graph g_0. First, we must ensure that the condition attached to the rule holds,*i.e.*, f must map the λ node in g to a node in g_0 that has exactly one pointer, the one from $f(n)$. Second, nodes n_1 and n_1' specify an additional redirection to be performed in the redirection phase (like the (λ-sp) rule). Thus, all pointers to $f(n_1)$ are redirected to the copy of n_1' We denote the the reduction by

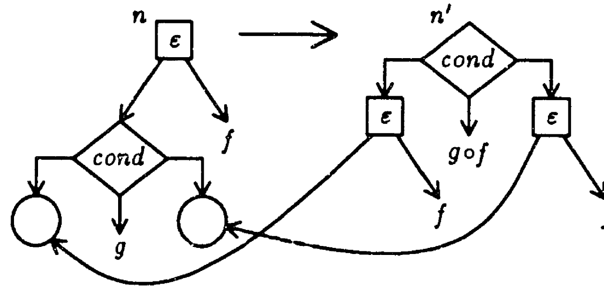
$$GC((g_0 +_f \beta_{fc})[f(n) := n''][f(n_1) := n_1'])$$

Rules to make subgraphs reducible

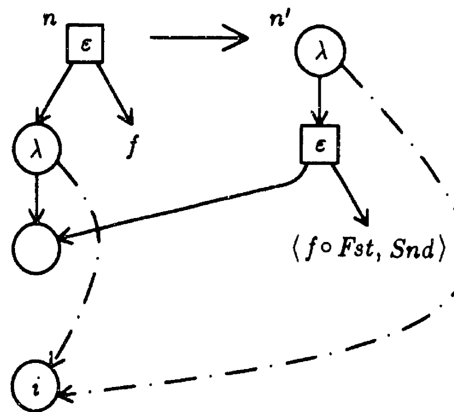
Figure 4.13 shows the three rules introduced in Section 3.7. As argued in that section, these rules preserve both translation and optimality.

A couple of points about the rule (ε-λ). First, the function on the right hand side is now $\langle f \circ Fst, Snd \rangle$, which reflects the fact that the structure of control environments is the reverse of what it used to be. Second, like the rule (β_{fc}), there is a condition attached to the rule. This rule is similar to the rule $(D\Lambda)$ in Curien's CCL.

(ε-c):



(ε-λ):



(Ap-c):

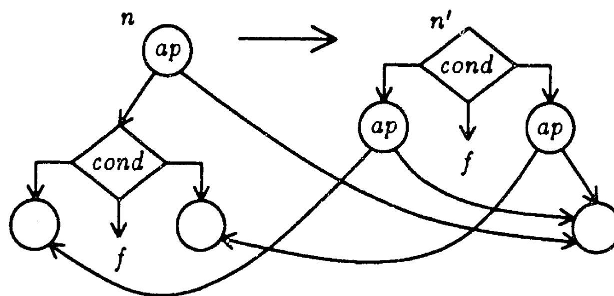


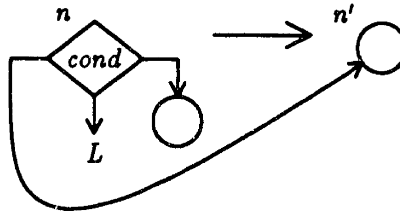
Figure 4.13: Rules $(\varepsilon\text{-}\lambda)$, $(\varepsilon\text{-}c)$ and $(Ap\text{-}c)$

Rules to make the computation rule simple

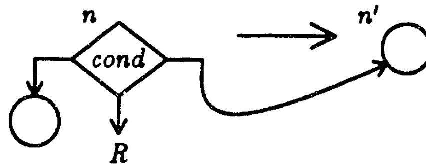
We introduce several new reduction rules, which are shown in Figures 4.14 and 4.15. They all have the form (g, n, n'). Intuitively, all these rules preserve translation as they correspond to various clauses or combination of clauses in the translation procedure.

The four rules shown in Figure 4.14 eliminate conditional nodes from graphs, and they obviously preserve optimality.

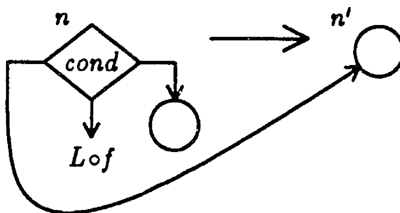
(C-L):



(C-R):



(C-L'):



(C-R'):

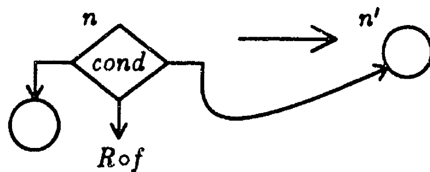
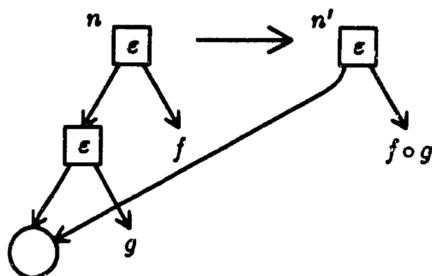


Figure 4.14: Rules to eliminate conditional nodes

The rule (ε-ε) shown in Figure 4.15 combines two ε nodes into one. Again, it obviously preserves optimality as it deals only with ε nodes.

The last rule we introduce in this section is a rule to push an ε node past an *ap* node. The rule, called (ε-ap) is shown in Figure 4.15. Unfortunately, the rule doesn't preserve optimality, since it may copy an *ap* node that should not be copied. Consider, for example, the application of this rule to a a graph so that the *ap* node on the left hand side of the rule maps to a shared

(ε-ε):



(ε-ap):

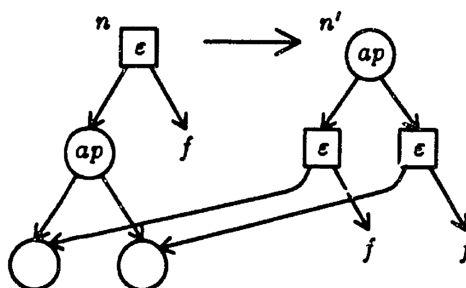


Figure 4.15: Rules (ε-ε), (ε-ap)

redex. The application of this rule will copy the shared redex, thus destroying optimality.

Therefore, one possibility is to simply exclude such a rule. However, some sort of a rule to push ε nodes past *ap* nodes is necessary if we want a simple strategy for applying rules. Without such a rule, it is not possible to eliminate conditional nodes that occur on the argument sides of applications. Conditional nodes on the function sides don't cause problem as they can always be pushed towards the root using the (Ap-c) rule. Now, if we can ensure that applications represented by the *ap* node are neither redexes nor can become redexes, then it is safe to apply the rule because applications represented by the *ap* node can never become shared redexes. The reduction strategy used by the interpreter will be such that the (ε-ap) rule is applied only when the above condition can be ensured (see next section).

We end the description of reduction rules with the following definition and and some comments.

Definition 4.23

$$E_g = (\varepsilon\text{-}\varepsilon), (\varepsilon\text{-ap}), (\varepsilon\text{-c}), (\varepsilon\text{-}\lambda), (\text{Ap-c})\ (\text{C-L}), (\text{C-R}), (\text{C-L'}), (\text{C-R'})$$
$$I = \{(\lambda\text{-sp}), (\beta_{fc})\} \cup E_g \cup F_g$$

The reduction rules used by the interpreter are the ones in the set I. The reduction rules in F_g and E_g with (ε-λ) applied without any restriction perform essentially the same function as the translation procedure and can be used to translate graphs.

4.4.5 Reduction Strategy

Before we proceed, we make some observations about leftmost reduction in the λ-calculus. Suppose M is a λ-term that has normal form. Then, the leftmost reduction of M to normal form can be decomposed into several reductions. First comes the leftmost reduction of M to a term N where N is in head normal form, *i.e.*, is of the form

$$\lambda_1 \ldots \lambda_n x A_1 \ldots A_m$$

for $n, m \leq 0$. Then, comes the leftmost reduction of A_1 to normal form, A_2 to normal form, and so on. Now, consider the term N. Any application to the left of the head variable x is neither a redex nor can become a redex.

As pointed out in the last section, the rule (ε-ap) should be applied only when applications represented by the *ap* node that is part of the redex are neither (β) redexes nor can become redexes. We use the above observation to design a reduction strategy that satisfies this condition.

1. Given a graph G that represents a De Bruijn term, the interpreter applies all the rules in I except the (ε-ap) rule in "leftmost" order to reduce G to a graph whose tree representation is of the form:

   $$\lambda_1 \ldots \lambda_n \underbrace{\text{only } \varepsilon \text{and ap's} \ldots}\, i \ldots$$

   and to the left of i there are no redexes other than (ε-ap) redexes.

   The leftmost redex in a graph is the first redex encountered in traversing the graph respecting the order of successors of a node. Note that the first successor of conditional node is the function part of the node. Thus, the interpreter first reduces the function part of a conditional node and then eliminates the conditional node using one of the four rules to remove conditional nodes.

2. Then, the interpreter reduces all those (ε-ap) redexes that correspond to the (ε-ap) redexes to the left of i in leftmost order.

3. Then, the interpreter applies steps (1) and (2) to subgraphs that are argument parts of *ap*'s to the left of i.

In the first step, if a graph represents a De Bruijn term, then it is indeed possible to reduce the graph to a graph whose tree representation is of the above form. To the left of i, an ε cannot be followed by either a λ or a *cond* as such a combination corresponds to a redex and can be reduced. Similarly, an *ap* cannot be followed by λ or *cond.* So ε and *ap* can be followed only by each other. They cannot be preceded by *cond* because then the *cond* cannot be discharged, and G cannot represent a De Bruijn term. So ε and *ap* can only be preceded by λ's.

Reductions in the second step don't destroy optimality because we claim that all applications represented by the *ap* node cannot become β_{fc} redexes (they certainly are not redexes). The reason is as follows. Suppose

$$ap_1(\ldots i_1, \ldots) \quad \ldots \quad ap_j(\ldots i_j, \ldots)$$

are all the application represented by the *ap* node. One of them, say ap_1 must be to the left of "head" i. Certainly, ap_1 cannot become a β_{fc} redex. We argue that none of them can become β_{fc} redexes. All of $i_1 \ldots i_j$ correspond to the same node in the graph. The following two requirements on graphs that represent De Bruijn terms imply that either all of them are free or all of them are bound: First, an i node has a binding pointer from only one λ node; second, the binding relation described using binding pointers is consistent with the binding relation in the De Bruijn term represented by the graph. If all of $i_1, \ldots, i_j$ are free, then none of the applications can become redexes. If they are all bound, then we claim they are all bound by a λ in $\lambda_1 \ldots \lambda_n$. i_1 is certainly bound by a λ in $\lambda_1 \ldots \lambda_n$, say λ_j. Suppose one of the other $i_2, \ldots i_j$ is bound by another λ-abstraction, say λ_o. By the two requirements on graphs mentioned above, both λ_j and λ_o correspond to the same λ node in the graph. Consider the (minimal) path from the root of the graph to this node. The path consists of only λ nodes, and one of them must have two pointers. But then the rule (λ-sp) is applicable.

Overall, the interpreter operates as follows:

1. Given a λ-expression in De Bruijn's notation, it constructs the initial graph representation of the expression.

2. It reduces the graph to normal form in the way described above.

3. It translates the result into a De Bruijn expression in a very simple way as described at the end of Section 4.4.3.

Optimizations

The two rules (β_{fc}) and (ε-λ) can be applied only if the λ node in an instance of these rules has exactly one pointer. One way to satisfy the condition is to simply precede an application of one of these rules by an application of the rule (λ-sp). Another way to satisfy the condition is to keep track of the number of pointers to a λ node, and apply (λ-sp) only if necessary. Both have their advantages and disadvantages. We follow the first approach for the simple reason that the interpreter doesn't have to maintain any extra information.

If (λ-sp) is always applied before applying the other two rules, then it is more efficient to have rules that combine(λ-sp) with the other two rules. Thus, we introduce the following combinations: (β_g) that combines (λ-sp) and (β_{fc}), and (ε-λ_g) that combines (λ-sp) and (ε-λ). These rules are shown in Figure 4.16. The notation used in describing these rules is similar to the notation used for (λ-sp) and (β_{fc}). We write them in the form

$$(g, n, n', n_1, n'_1, n_2, n'_2)$$

A occurrence of these rules in a graph is identified by assuming that they are of the form (g, n, n'). For the build phase the set of new nodes is given by

$$N_{new} = (N_{g|n'} \cup N_{g|n'_1}) - N_{g|n}.$$

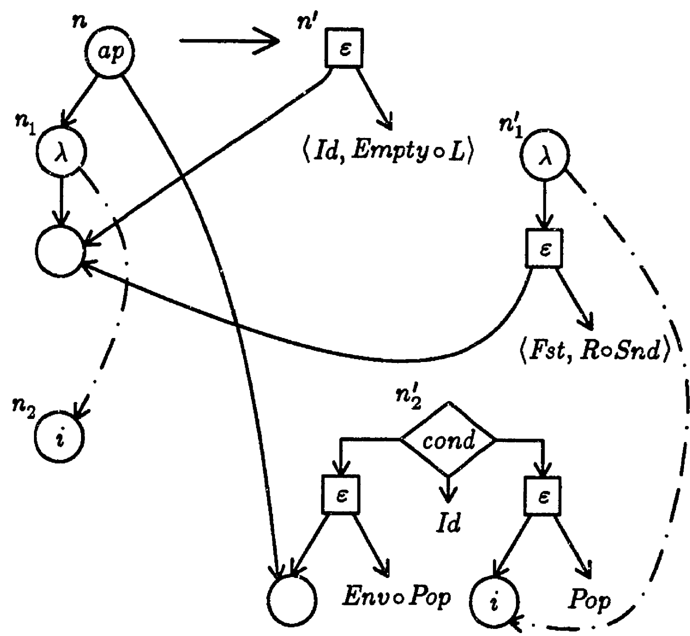
The three pairs of nodes, *i.e.*, a node and its primed version, specify three redirections to be performed in the redirection phase.

We introduce one more rule that optimizes the frequently occurring sequence (λ-sp), (ε-λ) and (β_{fc}). The rule, called (β'_g), is shown in Figure 4.17 using the notation given above.

The three rules discussed in this section can be used in place of rules β_{fc}, (λ-sp), and (ε-λ).

4.4.6 Summary

In this section, we gave a complete description of the interpreter. In Section 4.4.1, we described described functions and reduction rules for functions. We defined λ_{fc} graph in Section 4.4.2 and identified a subset of these graphs, called λ_{fc} graphs that represent De Bruijn terms, in Section 4.4.3. The interpreter operates on $\lambda_{fc\text{-}db}$ graphs. In Section 4.4.4, we described the remaining reduction rules. In Section 4.4.4, we gave the computation rule used by the interpreter. Finally, in Section 4.4.3, we provided translations between λ-expressions in DeBruijn's notation and $\lambda_{fc\text{-}db}$ graphs.



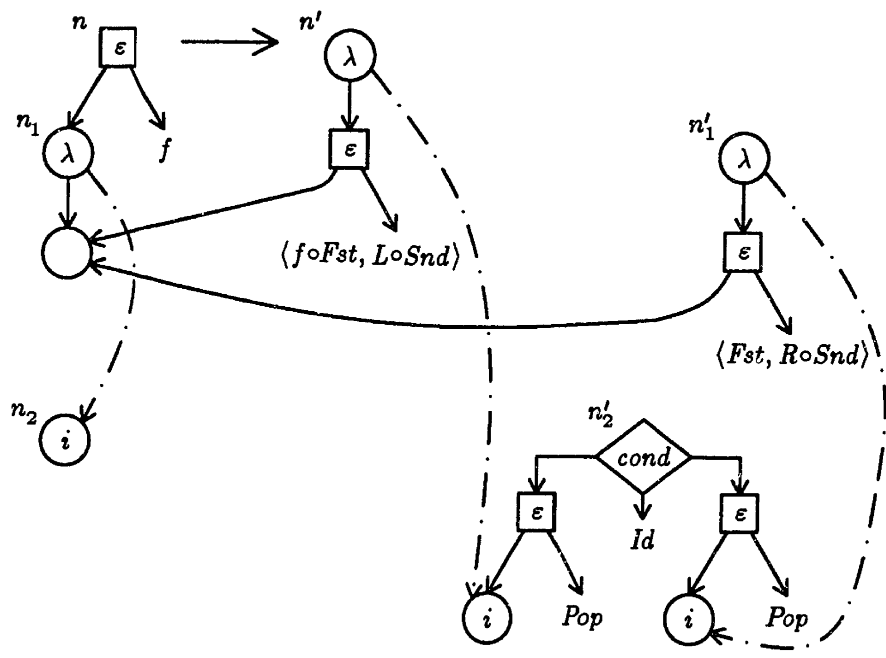


Figure 4.16: Rules (β_g) and (ε-λ_g)

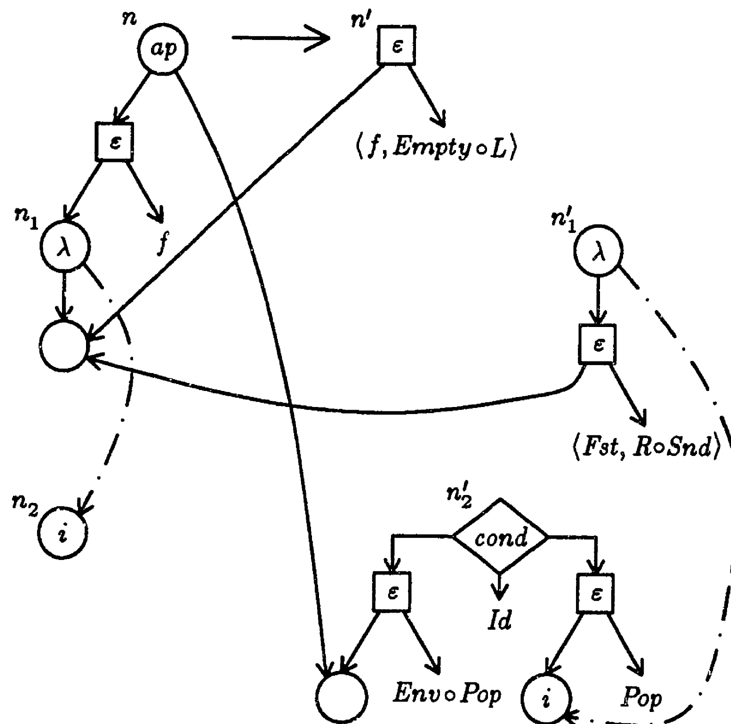


Figure 4.17: Rule (β'_g)

Chapter 5

λ_f Calculus

In this chapter, we describe a variation of De Bruijn notation, called λ_f calculus. The main advantage of this system over De Bruijn notation (or λ-calculus) is that changing of numbers (or renaming of variables) need not be done at the time a β redex is reduced but can be done incrementally. This makes the system a more attractive basis for designing graph reduction interpreters for the λ-calculus than either De Bruijn notation or the λ-calculus (see Sections 2.4 and 4.1). Indeed, the calculus was motivated by the interpreter presented in this thesis and will be used as an intermediate step in proving the correctness of the interpreter.

5.1 Functions on control environments

Control environments don't appear explicitly in terms and reduction rules, only functions on control environments do. However, the knowledge of the structure of control environments is helpful in understanding what various rules do.

For the purpose of this chapter, control environments carry two types of values: De Bruijn numbers and other control environments.

Definition 5.1 The following grammar defines the structure of control environments.

$$\begin{array}{lll} \mathit{control\text{-}env} & ::= & () \mid (\mathit{control\text{-}env}, \mathit{value}) \\ \mathit{value} & ::= & n \in N \mid \mathit{control\text{-}env} \end{array}$$

where () denotes the empty list. **Cenv** denotes the set of control environments and **Val** denotes the set of values. Variables $\rho, \rho_1, \ldots$ range over control environments.

Definition 5.2 Functions on control environments are expressions built using the following signature:

- Id, Fst and Snd of arity 0
- $\circ$ and $\langle -, - \rangle$ of arity 2

In other words,

$$f \quad ::= \quad Id \mid Fst \mid Snd \mid \langle f, f \rangle \mid f \circ f$$

F will denote the set of functions on environments. Variables $f, g, h, \ldots$ range over **F**.

We follow the convention that the composition operation is right associative; for example $f \circ g \circ h$ means $f \circ (g \circ h)$. Note that this convention is the opposite of the one used for the application operation.

The definitions of these functions are as follows. Type variables α, β and γ denote either **Cenv** or **Val**.

$Id : \alpha \to \alpha$	$Id\, x = x$
$Fst : \mathbf{Cenv} \to \mathbf{Cenv}$	$Fst\,(\rho, val) = \rho$
$Snd : \mathbf{Cenv} \to \mathbf{Val}$	$Snd\,(\rho, val) = val$
$f \circ g : \alpha \to \beta$	$(f \circ g)\, x = f(g\, x)$
$\langle f, g \rangle : \alpha \to \mathbf{Cenv}$	$\langle f, g \rangle\, x = (f\, x, g\, x)$

In the last two cases, the types of f and g should be consistent with the definitions.

Definition 5.3 F is the following set of reduction rules. The rules follow easily from the definitions of these functions.

(Ass): $(f \circ g) \circ h \longrightarrow f \circ (g \circ h)$

(IdL): $Id \circ f \longrightarrow f$

(IdR): $f \circ Id \longrightarrow f$

(Fst): $Fst \circ \langle f, g \rangle \longrightarrow f$

(Snd): $Snd \circ \langle f, g \rangle \longrightarrow g$

(Dpair): $\langle f, g \rangle \circ h \longrightarrow \langle f \circ h, g \circ h \rangle$

Theorem 5.4 *The system* $(\mathbf{F}, F)$ *is noetherian and confluent.*

Proof. The proof is straightforward but tedious. We use a polynomial interpretation to prove that the system is noetherian. To prove the system confluent, we show that the system is locally confluent. The confluence then follows from Newman's Lemma [17]. Note that the system is a

TRS. So, to prove local confluence, we simply check that local confluence holds for each pair of interfering rules [17].

Noetherian: We define a polynomial interpretation $(-)_\tau$ that maps terms in **F** to natural numbers and show that for any rule $l \longrightarrow r$ in F, $l_\tau > r_\tau$.

$$\begin{array}{lcl} Id_\tau, Fst_\tau, Snd_\tau & = & 2, \\ (f \circ g)_\tau & = & f_\tau g_\tau + f_\tau \\ \langle f, g\rangle_\tau & = & f_\tau + g_\tau + 1 \end{array}$$

We verify that the desired result holds for each rule.

1. (IdL), (IdR), (Fst), and (Snd): Straightforward as the right hand side of any of these rules is smaller (in size) than the left hand side.

2. (Ass):

$$\begin{array}{lclcl} l_\tau & = & ((f \circ g) \circ h)_\tau & = & (f_\tau g_\tau + f_\tau)h_\tau + f_\tau g_\tau + f_\tau \\ & = & f_\tau(g_\tau h_\tau + g_\tau) + f_\tau + f_\tau h_\tau & & \\ r_\tau & = & (f \circ (g \circ h))_\tau & = & f_\tau(g_\tau h_\tau + g_\tau) + f_\tau \end{array}$$

Thus, $l_\tau > r_\tau$.

3. (Dpair):

$$\begin{array}{lclcl} l_\tau & = & (\langle f, g\rangle \circ h)_\tau & = & (f_\tau + g_\tau + 1)h_\tau + (f_\tau + g_\tau + 1) \\ & = & (f_\tau h_\tau + f_\tau) + (g_\tau h_\tau + g_\tau) + 1 + h_\tau & & \\ r_\tau & = & (\langle f \circ h, g \circ h\rangle)_\tau & = & (f_\tau h_\tau + f_\tau) + (g_\tau h_\tau + g_\tau) + 1 \end{array}$$

Thus, $l_\tau > r_\tau$.

Thus, the system is noetherian.

Local confluence: We observe that (Ass) interferes with all the rules in F including itself, and (IdL) interferes with (IdR). The following diagrams show that local confluence holds for each pair of interfering rules.

1. (Ass) and (Ass):

$$\begin{array}{ccc} ((f \circ f') \circ g) \circ h & \xrightarrow[Ass]{} & (f \circ f') \circ (g \circ h) \\ {\scriptstyle Ass}\downarrow & & \downarrow{\scriptstyle Ass} \\ (f \circ (f' \circ g)) \circ h & \underset{Ass}{\twoheadrightarrow} & f \circ (f' \circ (g \circ h)) \end{array}$$

2. (Ass) and (IdL):

$$\begin{array}{ccc} (Id \circ f) \circ g & \xrightarrow[Ass]{} & Id \circ (f \circ g) \\ {\scriptstyle IdL}\downarrow & & {\scriptstyle IdL}\downarrow \\ f \circ g & \underset{\equiv}{- - - -} & f \circ g \end{array}$$

3. (Ass) and (IdR): There are two cases.

Case (i): The term in which the rules interfere is of the form $(f \circ g) \circ Id$.

$$\begin{array}{ccc} (f \circ g) \circ Id & \xrightarrow[Ass]{} & f \circ (g \circ Id) \\ {\scriptstyle IdR}\downarrow & & {\scriptstyle IdR}\downarrow \\ f \circ g & \underset{\equiv}{- - - -} & f \circ g \end{array}$$

Case (ii): The term in which the rules interfere is of the form $(f \circ Id) \circ g$.

$$\begin{array}{ccc} (f \circ Id) \circ g & \xrightarrow[Ass]{} & f \circ (Id \circ g) \\ {\scriptstyle IdR}\downarrow & & {\scriptstyle IdL}\downarrow \\ f \circ g & \underset{\equiv}{- - - -} & f \circ g \end{array}$$

4. (Ass) and (Fst):

$$\begin{array}{ccc} (Fst \circ \langle f,g \rangle) \circ h & \xrightarrow[Ass]{} & Fst \circ (\langle f,g \rangle \circ h) \\ {\scriptstyle Fst}\downarrow & & {\scriptstyle Dpair,\, Fst}\downarrow\downarrow \\ f \circ h & \underset{\equiv}{- - - - - -} & f \circ g \end{array}$$

5. (Ass) and (Snd):

$$\begin{array}{ccc} (Snd \circ \langle f,g \rangle) \circ h & \xrightarrow[Ass]{} & Snd \circ (\langle f,g \rangle \circ h) \\ {\scriptstyle Snd}\downarrow & & {\scriptstyle Dpair,\, Snd}\downarrow\downarrow \\ f \circ h & \underset{\equiv}{- - - - - -} & f \circ h \end{array}$$

6. (Ass) and (Dpair):

$$\begin{array}{ccc}
(\langle f, f' \rangle \circ g) \circ h & \xrightarrow[Ass]{} & \langle f, f' \rangle \circ (g \circ h) \\
\Big\downarrow {\scriptstyle Dpair} & & \Big\downarrow {\scriptstyle Dpair} \\
\langle f \circ g, f' \circ g \rangle \circ h & \underset{Dpair,\ Ass}{\twoheadrightarrow} & \langle f \circ (g \circ h), f' \circ (g \circ h) \rangle
\end{array}$$

7. (IdL) and (IdR):

$$\begin{array}{ccc}
Id \circ Id & \xrightarrow[IdL]{} & Id \\
\Big\downarrow {\scriptstyle IdR} & & \Big| {\scriptstyle \equiv} \\
Id & \underset{\equiv}{—\,—\,—} & Id
\end{array}$$

This proves that the system is locally confluent. □

In the rest of this section, we introduce some notation and prove some results, all of which will be useful in the subsequent sections. Proposition 5.5 will be used heavily in other proofs.

Notation:

- $f \circ g^n = f \underbrace{\circ g \circ \cdots \circ g}_{n \text{ times}}$. Note that $f \circ g^0 = f$. Similar comments apply for $f^n \circ g$.
- $n! = (Snd \circ Fst^n)$, for $n \geq 0$.
- For any function on environment f,

$$P^0(f) = f, \qquad P^n(f) = \langle P^{n-1}(f) \circ Fst, Snd \rangle$$

  We will write $P(f)$ for $P^1(f)$.

Note that the normal form of $P^n(f)$, for $n \leq 1$, is of the form

$$\langle f \circ Fst^n, (n-1)!\rangle, \ldots, 1!\rangle, 0!\rangle.$$

Proposition 5.5 *The following holds.*

1. $m! \circ P^n(f) \underset{F}{\twoheadrightarrow} m!$, *if* $m < n$.
2. $m! \circ P^n(f) \underset{F}{\twoheadrightarrow} Snd \circ Fst^{m-n} \circ f \circ Fst^n$, *if* $m \geq n$.

3. $P^n(f) \circ P^n(g) \xrightarrow[F]{} P^n(f \circ g)$.

4. *If* $f \circ P^n(g) \xrightarrow[F]{} m!$ *for* $0 \leq m < n$, *then* $f \xrightarrow[F]{} m!$.

Proof.

1 and 2. We show that

$$(Fst^m) \circ P^n(f) \xrightarrow[F]{} P^{n-m}(f) \circ Fst^m, \text{ for } 0 \leq m \leq n$$

Then, both parts follow easily. The proof of the above is by induction on m.

Base case: Suppose $m = 0$. The result is trivially true as both sides are equal to $P^n(f)$.

Induction step: Suppose the result is true for m. Then,

$$\begin{array}{lll} (Fst^{m+1}) \circ P^n(f) & & \\ \xrightarrow[Ass]{} & Fst \circ (Fst^m \circ P^n(f)) & \\ \xrightarrow[F]{} & Fst \circ (P^{n-m}(f) \circ Fst^m), & \text{by induction hypothesis,} \\ = & Fst \circ (\langle P^{n-m-1}(f) \circ Fst, Snd \rangle \circ Fst^m), & \text{by the definition of P,} \\ \xrightarrow[Dpair,\ Ass]{} & Fst \circ \langle P^{n-m-1}(f) \circ Fst^{m+1}, Snd \circ Fst^m \rangle & \\ \xrightarrow[Fst]{} & P^{n-(m+1)} \circ Fst^{m+1} & \end{array}$$

This completes the proof.

3. By induction on n.

Base case: Suppose $n = 0$. Then, both sides are simply $f \circ g$. So the result is trivially true.

Induction step: Suppose the result is true for n. Then,

$$\begin{array}{lll} P^{n+1}(f) \circ P^{n+1}(g) & & \\ = & \langle P^n(f) \circ Fst, Snd \rangle \circ \langle P^n(g) \circ Fst, Snd \rangle & \\ \xrightarrow[F]{} & \langle P^n(f) \circ P^n(g) \circ Fst, Snd \rangle & \\ \xrightarrow[F]{} & \langle P^n(f \circ g) \circ Fst, Snd \rangle & \text{by induction hypothesis,} \\ = & P^{n+1}(f \circ g). & \end{array}$$

Thus, the result is true.

4. By contradiction. Without loss of generality, we may assume that both f and $P^n(g)$ are in normal form and show that f must be $m!$. Note that the normal form of $P^n(g)$ is

$$\langle g \circ Fst^n, (n-1)!\rangle, \ldots, 1!\rangle, 0!\rangle.$$

If If $f \circ P^n(g) \underset{F}{\twoheadrightarrow} m!$, then f must first "select" an element from these nested pairs. We consider the following two cases.

Suppose f selects an element other than $g \circ Fst^n$. Then, f must either be $k!$ or $f' \circ k!$ for some k and f'. In either case, Proposition 5.5(1) implies that $f \circ P^n(f) \underset{F}{\twoheadrightarrow} f$. So if $f \not\equiv m!$, then $f \circ P^n(f)$ cannot reduce to $m!$.

Now, suppose that f selects the element $g \circ Fst^n$. Then, f must be of either Fst^n or $f' \circ Fst^n$ for some f'. We consider the second case; the proof for the first case is similar. Now, $f' \circ Fst^n \circ P^n(g) \underset{F}{\twoheadrightarrow} f' \circ g \circ Fst^n$. But $f' \circ g \circ Fst^n$ cannot reduce to $m!$ for $m < n$ no matter what f' and g are.

Thus, f must reduce to $m!$.

□

5.2 λ_f terms and E reduction rules

Definition 5.6 The syntax of terms of the λ_f calculus is as follows:

$$Term \ ::= \quad i \mid \lambda.Term \mid Term\,Term \mid \varepsilon(Term, f) \ \text{ where } \ f \in \mathbf{F}$$

The set of terms will be denoted by Λ_f. Variables $M, N, \ldots$ range over Λ_f.

We introduce reduction rules to combine ε terms and to propagate ε terms towards the leaves.

Definition 5.7 E is the following set of reduction rules:

(ε-ε): $\quad \varepsilon(\varepsilon(M, f), g) \longrightarrow \varepsilon(M, f \circ g)$

(ε-ap): $\quad \varepsilon(M\,N, f) \longrightarrow \varepsilon(M, f)\,\varepsilon(N, f)$

(ε-λ): $\quad \varepsilon(\lambda.M, f) \longrightarrow \lambda.\varepsilon(M, P(f))$

Theorem 5.8 *The system $(\Lambda_f, E \cup F)$ is noetherian and confluent.*

Proof. The proof proceeds as the proof of Theorem 5.4.

Noetherian: We extend the polynomial interpretation used in the proof of Theorem 5.4 to λ_f terms as follows:

$$\begin{aligned} \varepsilon(M,f)_\tau &= (M_\tau)^2 f_\tau \\ (M\,N)_\tau &= 2M_\tau + N_\tau \\ (\lambda.M)_\tau &= 5M_\tau \end{aligned}$$

To show that $l_\tau > r_\tau$ for any rule rule $l \longrightarrow r$ in $E \cup F$, we have to show it only for E rules. The proof of Theorem 5.4 shows that the result holds for F rules.

1. (ε-ε):

$$\begin{aligned} l_\tau &= (\varepsilon(\varepsilon(M,f),g))_\tau = ((M_\tau)^2 f_\tau)^2 g_\tau \\ &= (M_\tau)^2 f_\tau g_\tau (M_\tau)^2 f_\tau \\ r_\tau &= \varepsilon(M, f \circ g)_\tau = (M_\tau)^2 (f_\tau g_\tau + f_\tau) \\ &= (M_\tau)^2 f_\tau g_\tau + (M_\tau)^2 f_\tau \end{aligned}$$

Now $ab > a + b$ is true for any numbers $a, b > 2$. Then, $l_\tau > r_\tau$, since both $(M_\tau)^2 f_\tau g_\tau$ and $(M_\tau)^2 f_\tau$ are greater than 2.

2. (ε-ap):

$$\begin{aligned} l_\tau &= \varepsilon((M\,N),f)_\tau = (2M_\tau + N_\tau)^2 f_\tau \\ r_\tau &= (\varepsilon(M,f)\,\varepsilon(N,f))_\tau = 2(M_\tau)^2 f_\tau + (N_\tau)^2 f_\tau \end{aligned}$$

Thus, $l_\tau > r_\tau$.

3. (ε-λ):

$$\begin{aligned} l_\tau &= \varepsilon(\lambda.M,f)_\tau = (5M_\tau)^2 f_\tau \\ &= 15(M_\tau)^2 f_\tau + 10(M_\tau)^2 f_\tau \\ r_\tau &= (\lambda.\varepsilon(M, \langle f \circ Fst, Snd\rangle))_\tau = 5(M_\tau)^2(2f_\tau + f_\tau + 2 + 1) \\ &= 15(M_\tau)^2 f_\tau + 15(M_\tau)^2 \end{aligned}$$

Since $f_\tau \geq 2$, $10(M_\tau)^2 f_\tau > 15(M_\tau)^2$. Thus, $l_\tau > r_\tau$.

Thus, the system is noetherian.

Local confluence: We observe that the rules in F don't interfere with rules in E. Moreover, local confluence for F rules has already been shown in Theorem 5.4. Thus, we have to show

the result only for E rules. Note that the rule (ε-ε) interferes with all the rules in E including itself. The following diagrams show that local confluence holds for each pair of interfering rules.

1. (ε-ε) and (ε-ε):

$$\begin{array}{ccc}
\varepsilon(\varepsilon(\varepsilon(M,f),g),h) & \xrightarrow[\varepsilon\text{-}\varepsilon]{} & \varepsilon(\varepsilon(M,f),g\circ h) \\
\Big\downarrow{\scriptstyle \varepsilon\text{-}\varepsilon} & & \Big\downarrow{\scriptstyle \varepsilon\text{-}\varepsilon} \\
\varepsilon(\varepsilon(M,f\circ g),h) & \xrightarrow[\varepsilon\text{-}\varepsilon,Ass]{} & \varepsilon(M,f\circ(g\circ h))
\end{array}$$

2. (ε-ε) and (ε-ap):

$$\begin{array}{ccc}
\varepsilon(\varepsilon(M\,N,f),g) & \xrightarrow[\varepsilon\text{-}\varepsilon]{} & \varepsilon(M\,N,f\circ g) \\
\Big\downarrow{\scriptstyle \varepsilon\text{-}ap} & & \Big\downarrow{\scriptstyle \varepsilon\text{-}ap} \\
\varepsilon(\varepsilon(M,f)\,\varepsilon(N,f),g) & \xrightarrow[\varepsilon\text{-}ap,\varepsilon\text{-}\varepsilon]{} & \varepsilon(M,f\circ g)\,\varepsilon(N,f\circ g)
\end{array}$$

3. (ε-ε) and (ε-λ) :

$$\begin{array}{ccccc}
\varepsilon(\varepsilon(\lambda.M,f),g) & & \xrightarrow[\varepsilon\text{-}\varepsilon]{} & & \varepsilon(\lambda.M,f\circ g) \\
\Big\downarrow{\scriptstyle \varepsilon\text{-}\lambda} & & & & \Big\downarrow{\scriptstyle \varepsilon\text{-}\lambda,Ass} \\
\varepsilon(\lambda.\varepsilon(M,P(f)),g) & \xrightarrow[\varepsilon\text{-}\lambda]{} & \lambda.\varepsilon(\varepsilon(M,P(f)),P(g)) & \xrightarrow[\varepsilon\text{-}\varepsilon,F]{} & \lambda.\varepsilon(M,\langle f\circ(g\circ Fst),Snd\rangle)
\end{array}$$

Thus, the system is locally confluent. Since the system is noetherian, Newman's Lemma implies that the system is confluent. □

5.3 Translations between λ_f calculus and De Bruijn notation

In this section we define translations between terms in the λ_f calculus and λ-terms in De Bruijn notation.

Definition 5.9 For any term A in De Bruijn's notation, its translation into a term in the λ_f-calculus is denoted by A_{λ_f} and is defined by induction on the structure of A.

$$
\begin{aligned}
n_{\lambda_f} &= \varepsilon(i, n!) \\
(B\,C)_{\lambda_f} &= B_{\lambda_f} C_{\lambda_f} \\
(\lambda.B)_{\lambda_f} &= \lambda.B_{\lambda_f}
\end{aligned}
$$

One way to translate λ_f terms to terms in De Bruijn's notation (or λ-terms) is to code primitive functions on environments and ε as λ-expressions.

We take a different approach. We define a subset of Λ_f and define translation for only terms in this subset. The definition of this subset is based on the observation that, for any λ-term in De Bruijn notation, its translation into a term in the λ_f calculus has a special structure. First, it is in $E \cup F$ normal form. Second, ε and i occur only in the combination $\varepsilon(i, n!)$ for some n.

Definition 5.10 A λ_f term *represents a De Bruijn term* iff the $E \cup F$ normal form of the term satisfies the following property: ε and i occur only in the combination $\varepsilon(i, n!)$ for some n. In other words, the $E \cup F$ normal form of the term can be obtained by translating a λ-term in De Bruijn notation. The notion is well-defined as $E \cup F$ rules are noetherian and confluent.

Definition 5.11 For any term λ_f term M that represents a De Bruijn term, its translation into a λ-term in De Bruijn notation is denoted by M_{DB} and is defined in the obvious way—apply inverse of $(-)_{\lambda_f}$ to $E \cup F$ normal form of M.

Intuitively, the set of terms that represent De Bruijn terms can be partitioned into equivalence classes. Each equivalence class consists of terms that have the same $E \cup F$ normal form. The term in $E \cup F$ normal form in an equivalence class can be considered as the canonical representative of the class. Terms in De Bruijn's notation are in 1-1 correspondence with these equivalence classes. The β_f rule of λ_f calculus will transform a term in one equivalence class to a term in another equivalence class.

The following proposition shows that if a λ_f term represents a De Bruijn term, then certain types of subexpressions of the term also represent De Bruijn terms.

Proposition 5.12 *Let $C[M]$ be a λ_f term that represents a De Bruijn term. Suppose the context C contains no ε-term of which M is a subterm, and furthermore, suppose that $M \not\equiv i$. Then, M also represents a De Bruijn term.*

Proof. Since M is not a subterm of an ε-term, M cannot be a function on environments. Moreover, E reductions don't propagate from the context C to M. Thus, the $E \cup F$ reduction of $C[M]$ to normal form can be decomposed as follows:

$$C[M] \overset{in\ C}{\twoheadrightarrow} C'[M] \overset{in\ M}{\twoheadrightarrow} C'[M']$$

The conditions imposed on M imply that M', which is the $E \cup F$ normal form of M, is an abstraction, an application or an ε-term. Since $C[M]$ represents a De Bruijn term, $C'[M']$ also represents a De Bruijn term. It follows from the definition that M' represents a De Bruijn term. Since $M \underset{E,F}{\twoheadrightarrow} M'$, M' is also represents a De Bruijn term. □

5.4 Traces of subexpressions relative to $E \cup F$ reduction

Let $M \underset{E,F}{\twoheadrightarrow} N$. For an application, λ-abstraction or i in M, we define the notion of its trace in N relative the the given reduction as follows. Underline the first symbol of the subexpression and carry the underlining through reduction in the usual way except for the following two cases:

$(\varepsilon\text{-ap})$: $\quad \varepsilon(\underline{(}M\ N), f) \longrightarrow \underline{(}\varepsilon(M, f)\ \varepsilon(N, f))$

$(\varepsilon\text{-}\lambda)$: $\quad \varepsilon(\underline{\lambda}.M, f) \longrightarrow \underline{\lambda}.\varepsilon(M, F(f))$

Proposition 5.13 *Let M be a λ_f term, and let N be its $E \cup F$ normal form. Then, any application, λ-abstraction, and i in M has exactly one trace in N and the trace is independent of the reduction from M to N.*

Proof. The uniqueness is proved by inspection of the rules and the definition of traces. To show that trace of a subexpression is independent of the particular choice or reduction to normal form, we can show that confluence holds even after underlining. The proof is similar to the proof of confluence in Theorem 5.8 and is omitted. □

5.5 Free and bound occurrences of i

An occurrence of i in a term acts like a variable, and it is either free in the term or bound by some λ-abstraction in the term. As in the λ-calculus, it is not necessary to define the notions of of free and bound occurrences. All that is needed is an explicit definition of substitution operation, which we will give in the next section. However, these notions prove useful in discussions.

Basically, the notions of free and bound occurrences of i in a λ_f term are derived from the corresponding notions in De Bruijn notation, keeping the translations between two systems in mind. The following definition is phrased the way we want to phrase the substitution operation in the next section.

Definition 5.14 Let $\lambda.M$ be an abstraction in the λ_f calculus. Occurrences of i bound by this abstraction are determined as follows. Consider an occurrence of i in $\lambda.M$, and suppose there are n λ-abstractions, not counting the outermost λ-abstraction, that enclose the occurrence of i. Now, consider the trace of the occurrence of i in the $E \cup F$normal form of the expression. The occurrence of i in the original expression is bound by the outermost λ-abstraction iff, in the normal form, the trace of the occurrence of i occurs in the form $\varepsilon(i, n!)$.

An occurrence of i in a λ_f term is *bound* iff it is bound by a λ-abstraction inside the term; otherwise, it is *free*.

It is not obvious that this is a good definition in the sense that a bound occurrence of i in a λ_f term is bound by exactly one λ-abstraction. However, this is indeed the case as shown by the following proposition.

Proposition 5.15 *A bound occurrence of i in a λ_f term is bound by exactly one λ-abstraction.*

Proof. By contradiction. Suppose there are two distinct λ-abstractions that bind an occurrence of i. Then, one of them must be part of the other. Consider the inner λ-abstraction, say $\lambda.N$. Suppose there are n λ-abstractions inside N that enclose the occurrence of i. Let $\lambda.N'$ be the $E \cup F$ normal form of $\lambda.N$. Then, in $\lambda.N'$, the trace of the occurrence i must occur in the form $\varepsilon(i, n!)$.

Now, consider the $E \cup F$ normal form of the outer λ-abstraction. First, we show that the trace of the occurrence of i must occur in the form $\varepsilon(i, n!)$. The outer abstraction is of the form $\lambda.C[\lambda.N]$ where $C[]$ is a context. Since $E \cup F$ rules are confluent, we may assume that the reduction to normal form is of the following form:

$$C[\lambda.N] \overset{in\ C}{\twoheadrightarrow} C'[\lambda.N] \overset{in\ N}{\twoheadrightarrow} C'[\lambda.N'] \twoheadrightarrow M'$$

Now, $\lambda.N'$ is the $E \cup F$ normal form of $\lambda.N$. Thus, in $C'[\lambda.N']$, the trace of the occurrence of i must occur in the form $\varepsilon(i, n!)$ (see the first paragraph). Now, we consider two cases. First, suppose $C'[\lambda.N']$ is in $E \cup F$ normal form. Then, the desired result holds. Otherwise, $C'[\lambda.N']$ must be of the form

$$C'[\lambda.N'] \equiv C''[\varepsilon(\lambda.N', f)].$$

Moreover, $C''[]$ is in $E \cup F$ normal form and contains no ε that encloses the expression in the hole. Let M' be the $E \cup F$ normal form of $\varepsilon(\lambda.N', f)$. Then, $M \equiv C''[M']$. In M, suppose

the trace of the occurrence of i occurs in the form $\varepsilon(i,g)$. Then, g must be the F normal form of $n! \circ P^{n+1}(f)$. The reason is this: The reduction of $\varepsilon(\lambda.N', f)$ to M' simply consists of propagating the ε to the leaves, and the ε crosses $(n+1)$ λ-abstractions before it reaches the trace of the occurrence of i and doesn't combine with any other ε except at the very end. Now, by Proposition 5.5(1), $n! \circ P^{n+1}(f) \underset{F}{\twoheadrightarrow} n!$. Thus, $g \equiv n!$.

To complete the proof, we note that, in the body of the outer abstraction, there are at least $(n+1)$ λ-abstractions that enclose the occurrence of i. Thus, the occurrence of i cannot be bound by the outer λ-abstraction. □

5.6 β-reduction

In this section, we define the β_f rule that simulates β-reduction in De Bruijn notation. A β_f-redex is a λ_f-term of the form $(\lambda.M)\,N$. Its reduction involves *naively* substituting N for each occurrence of i in M that is bound by the λ-abstraction and modifying the *root* of the redex.

Definition 5.16

$$(\beta_f): \qquad (\lambda.M)\,N \longrightarrow \varepsilon(M[N], \langle Id, Id\rangle)$$

where $M[N]$ is the term obtained by naively substituting N for all occurrences of i in M that are bound by the abstraction $\lambda.M$. More explicitly,

$$M[N] = Subst[M, N, 0, Id],$$

where

$$\begin{aligned}
Subst[i, N, n, f] &= \begin{cases} N & \text{if } f \underset{F}{\twoheadrightarrow} n! \\ i & \text{otherwise} \end{cases} \\
Subst[(P\,Q), N, n, f] &= Subst[P, N, n, f]\; Subst[Q, N, n, f] \\
Subst[\lambda.P, N, n, f] &= \lambda.(Subst[P, N, n+1, P(f)]) \\
Subst[\varepsilon(P, g), N, n, f] &= \varepsilon(Subst[P, N, n, g \circ f], f)
\end{aligned}$$

Note that *Subst* simply replaces certain occurrences of i in M by N and doesn't modify either M or N in any other way. The parameters n and f in the definition of *Subst* are used to determine the occurrences of i that must be replaced by N (*cf.* the definition of bound occurrences). Also note that it is not necessary to define free and bound occurrences; all that is needed is the definition of the substitution operation given above.

We will also use the following derived rule:

(β'_f): $\quad \varepsilon(\lambda.M, f)\, N \longrightarrow \varepsilon(M[N], \langle f, Id\rangle)$

The notion of trace of an application, λ-abstraction or an i relative to a β_f reduction is defined by carrying the underlining through the reduction in the obvious way.

5.7 β_f reduction commutes with $E \cup F$ reductions

The main result in this section is Theorem 5.17, which proves a commutativity property of β_f reduction. The result will be used to show the equivalence of λ_f calculus and De Bruijn notation. The result hold only for terms that represent De Bruijn terms. The reason for this as well as the reason for the complexity of the proofs is that (ε-ap) reduction interferes with β_f reduction, that is, the term $\varepsilon(\lambda.M, f)\, N$ can be reduced in two ways. For a general λ_f terms, these two reducts cannot be reduced to a common term. Since the main part of the β_f reduction is the substitution operation, most of the propositions prove properties of this operation.

Theorem 5.17 *Let M be λ_f term that represents a De Bruijn term, and let N be its $E \cup F$ normal form. Then, the following diagram commutes.*

$$\begin{array}{ccccc} M & & \xrightarrow{\quad\beta_f\quad} & & M' \\ \Big\downarrow{\scriptstyle E,F} & & & & \Big\downarrow{\scriptstyle E,F} \\ N & \xrightarrow[\beta_f]{} & N'' & \xrightarrow[E,F]{} & N' \end{array}$$

Proof of this theorem will be given after a series of Lemmas. First, we prove the theorem for some simple cases.

Definition 5.18 In a λ_f-term, a β_f redex and an ε-*ap* redex *interfere* iff they occur in the following form:

$$\underbrace{\varepsilon(\underbrace{(\lambda.M)\, N}_{\beta_f \text{ redex}}, f)}_{\varepsilon-ap \text{ redex}}$$

Lemma 5.19 *Let M be a λ_f-term. Then, the following diagram commutes provided the $E \cup F$ redex doesn't interfere with the β_f redex and is not inside the abstraction part of the β_f redex.*

$$\begin{array}{ccc} M & \xrightarrow[\beta_f]{} & M' \\ {\scriptstyle E,F}\downarrow & & \downarrow{\scriptstyle E,F} \\ N & \dashrightarrow_{\beta_f} & N' \end{array}$$

Proof. By case analysis on the relative positions of the two redexes. In M, let R be the contracted $E \cup F$ redex, and and let B be the contracted β_f redex.

1. R and B are disjoint. Then $M \equiv C[R, B]$ for some context $C[,]$ with two holes. Let where R' and B' be the reducts of E and B respectively. Then, we have

$$\begin{array}{ccc} C[R,B] & \xrightarrow[\beta_f]{} & C[R,B'] \\ {\scriptstyle E,F}\downarrow & & \downarrow{\scriptstyle E,F} \\ C[R',B] & \dashrightarrow_{\beta_f} & C[R',B'] \end{array}$$

2. B is a subexpression of R. Then, R must be an E redex. We consider the following three cases based on the E rule applied.

   (a) The rule applied is (ε-λ): Then, $M \equiv C[\varepsilon(\lambda.S, f)]$ for some context $C[]$, and B must be inside S. Suppose $S \xrightarrow[\beta_f]{B} S'$. Then, we have the following diagram.

$$\begin{array}{ccc} C[\varepsilon(\lambda.S,f)] & \xrightarrow[\beta_f]{} & C[\varepsilon(\lambda.S',f)] \\ {\scriptstyle E,F}\downarrow & & \downarrow{\scriptstyle E,F} \\ C[\lambda.\varepsilon(S,P(f))] & \dashrightarrow_{\beta_f} & C[\lambda.\varepsilon(S',P(f))] \end{array}$$

   (b) The rule applied is (ε-ap). Then, $M \equiv C[\varepsilon(S_1\, S_2, f)]$. Since R and B don't interfere, B must be either inside S_1 or inside S_2. The rest is trivial to verify by inspection; see, for example, (ε-λ) rule.

   (c) The rule applied is (ε-ε). Then, $M \equiv C[\varepsilon(\varepsilon(S, f), g)]$, and B must be inside S. Again, the rest is easy to verify by inspection.

3. R is a subexpression of B. Let $M \equiv C[(\lambda.S)\,T]$. Then, B must be inside T. Suppose $T \xrightarrow[E,F]{R} T'$. We write $(\lambda.S)$ as $\lambda.S[i,\ldots,i]$ where the displayed occurrences of i are the ones bound by the λ-abstraction. Then,

$$\begin{array}{ccc}
C[(\lambda.S[i,\ldots,i])\,T] & \xrightarrow[\beta_f]{} & C[\varepsilon(S[T,\ldots,T],\langle Id,Id\rangle)] \\
\Big\downarrow{\scriptstyle E,F} & & \Big\downarrow\!\!\!\downarrow{\scriptstyle E,F} \\
C[(\lambda.S[i,\ldots,i])\,T'] & \dashrightarrow_{\beta_f} & C[\varepsilon(S[T',\ldots,T'],\langle Id,Id\rangle)]
\end{array}$$

Note that the reduction on the right hand side reduces a set of disjoint redexes, and thus, the order in which the redexes are reduced is not important.

This completes the proof of the lemma. □

The remaining two cases are: the $E \cup F$ redex is inside the abstraction part of the β_f redex, and the $E \cup F$ redex interferes with the β_f redex. Lemma 5.22 can be used to show the result in the first case. Lemma 5.19 shows the result in the latter case.

The next two lemmas show some properties of the substitution operation.

Lemma 5.20 *If $f \twoheadrightarrow_F f'$, then $Subst[M,N,n,f] = Subst[M,N,n,f']$.*

Proof. By induction on the structure of M. Suppose $M \equiv i$. Since F rules are confluent, f reduces to f' implies that they have the same normal form. Then,

$$\begin{aligned}
Subst[i,N,n,f] &= \begin{cases} N & \text{if } f \twoheadrightarrow_F n! \\ i & \text{otherwise} \end{cases} \\
&= Subst[i,N,n,f'].
\end{aligned}$$

The other cases follow easily from the induction hypothesis and the definition of the substitution operation. □

Lemma 5.21 *Let M and N be λ_f terms. Then,*

$$Subst[M,N,0,Id] \;=\; Subst[M,N,0,P(f)]$$

for any function on environments f.

Proof. By induction on the structure of M, we show that

$$Subst[M,N,n,g \circ P^n(Id)] \;=\; Subst[M,N,n,g \circ P^n(P(f))]$$

for any f and g. Then, to prove the result, we take $g = Id$, $n = 0$ and use Lemma 5.20.

1. $M \equiv i$: First, we show that

$$g \circ P^n(Id) \underset{F}{\twoheadrightarrow} n! \Longleftrightarrow g \circ P^n(P(f)) \underset{F}{\twoheadrightarrow} n!$$

($\Rightarrow$) In this direction, we have

$$\begin{aligned} g \circ P^n(P(f)) &=_F g \circ P^n(Id) \circ P^n(P(f)), && \text{by Proposition 5.5(3)} \\ &=_F n! \circ P^n(P(f)), && \text{by assumption} \\ &=_F Snd \circ P(f) \circ Fst^n, && \text{by Proposition 5.5(2)} \\ &=_F n! \end{aligned}$$

($\Leftarrow$) Now, $P^n(P(f)) = P^{n+1}(f)$. Since $g \circ P^n(P(f)) \underset{F}{\twoheadrightarrow} n!$, Proposition 5.5(4) implies that $g \underset{F}{\twoheadrightarrow} n!$. Then, Proposition 5.5(3) implies that

$$\begin{aligned} g \circ P^n(Id) &\underset{F}{\twoheadrightarrow} Snd \circ Id \circ Fst^n, \quad \text{which} \\ &\underset{F}{\longrightarrow} n! \end{aligned}$$

Using the above, we have

$$\begin{aligned} Subst[i, N, n, g \circ P^n(Id)] &= \begin{cases} N & \text{if } g \circ P^n(Id) \underset{F}{\twoheadrightarrow} n! \\ i & \text{otherwise} \end{cases} \\ &= Subst[i, N, n, g \circ P^n(P(f))]. \end{aligned}$$

2. $M \equiv \lambda.R$: Then,

$$\begin{aligned} LHS &=_{def} \lambda.Subst[R, N, n+1, P(g \circ P^n(Id))] \\ &= \lambda.Subst[R, N, n+1, P(g) \circ P^n(Id))] \\ RHS &=_{def} \lambda.Subst[R, N, n+1, P(g \circ P^n(P(f)))] \\ &= \lambda.Subst[R, N, n+1, P(g) \circ P^n(P(f))]. \end{aligned}$$

The last equality in both cases is derived using Propositions 5.5(3) and 5.20. The result then follows from the induction hypothesis.

3. $M \equiv R\,S$: In this case, the result follows easily from the induction hypothesis and the definition of the substitution operation.

4. $M \equiv \varepsilon(R, h)$: Again, the result follows easily from the induction hypothesis and the definition of the substitution operation.

□

The above lemma also shows that the (β'_f) rule is implied by the other rules. The next lemma shows that the substitution operation commutes with $E \cup F$ reductions. It will be used in the case when $E \cup F$ redexes are inside the abstraction part of the β_f redex.

Lemma 5.22 *Let M be a λ_f-term. Then,*

$$M \underset{E,F}{\longrightarrow} M' \;\Rightarrow\; M[N] \underset{E,F}{\longrightarrow} M'[N]$$

Proof. By induction on the structure of M, we show that

$$M \underset{E,F}{\longrightarrow} M' \;\Rightarrow\; Subst[M, N, n, f] \underset{E,F}{\longrightarrow} Subst[M', N, n, f]$$

1. $M \equiv i$. The result is vacuously true since M is in $E \cup F$ normal form.

2. $M \equiv R\,S$. The redex reduced in the reduction $M \longrightarrow M'$ must be either in R or in S. Thus, either $M' \equiv R'\,S$ and $R \underset{E,F}{\longrightarrow} R'$, or $M' \equiv R\,S'$ and $S \underset{E,F}{\longrightarrow} S'$. In both cases the result follows from the induction hypothesis and the definition of substitution operation.

3. $M \equiv \lambda.R$. The redex must be in R. Thus, $M' \equiv (\lambda.R')$ and $R \underset{E,F}{\longrightarrow} R'$. Again, the result follows from the hypothesis and the definition of substitution operation.

4. $M \equiv \varepsilon(R, g)$. We consider the following three cases.

   (a) The redex reduced is in R. Then, the result follows from the induction hypothesis (see the earlier cases).

   (b) The redex reduced is in g. Then, the result follows from Lemma 5.20.

   (c) The redex reduced is $\varepsilon(R, g)$ itself. Then, we consider three cases based on type of the reduction rule applied. Let LHS denote $Subst[M, N, n, f]$, and let RHS denote $Subst[M', N, n, f]$.

   (i) The rule applied is $(\varepsilon\text{-}\lambda)$. Then, $M \equiv \varepsilon(\lambda.S, g)$ and

$$\begin{aligned} LHS &=_{def} \varepsilon(\lambda.Subst[S, N, n+1, P(g \circ f)], g) \\ &= \varepsilon(\lambda.Subst[S, N, n+1, P(g) \circ P(f)], g), \\ &\qquad \text{by Propositions 5.5(3) and 5.20} \\ &\underset{\varepsilon-\lambda}{\longrightarrow} \lambda.\varepsilon(Subst[S, N, n+1, P(f) \circ P(g)], P(f)) \\ &=_{def} RHS. \end{aligned}$$

(ii) The rule applied is (ε-λ). Then, $M \equiv \varepsilon(\varepsilon(S,h),g)$, and

$$\begin{array}{lll} LHS & =_{def} & \varepsilon(\varepsilon(Subst[S,N,n,h\circ g\circ f],h),g) \\ & = & \varepsilon(\varepsilon(Subst[S,N,n,(h\circ g)\circ f],h),g), \quad \text{by Proposition 5.20} \\ & \xrightarrow[\varepsilon-\varepsilon]{} & \varepsilon(Subst[S,N,n,(h\circ g)\circ f],h\circ g), \\ & =_{def} & RHS \end{array}$$

(iii) The rule applied is (ε-ε). Then, $M \equiv \varepsilon(S_1\, S_2, g)$, and

$$\begin{array}{lll} LHS & =_{def} & \varepsilon(Subst[S_1,N,n,g\circ f]\; Subst[S_2,N,n,g\circ f],g) \\ & \xrightarrow[\varepsilon-ap]{} & \varepsilon(Subst[S_1,N,n,g\circ f],g)\; \varepsilon(Subst[S_2,N,n,g\circ f],g) \\ & =_{def} & RHS. \end{array}$$

This completes the proof of the lemma. □

The next lemma is the most interesting one. Intuitively, it shows that the information needed by the argument part of a β_f redex either can be made part of the argument part directly or can be supplied later. Note that the lemma holds only for λ_f terms that represent De Bruijn terms. Also note that the restriction that the λ_f term be in $E \cup F$ normal form is not necessary; it is there to simplify the proof.

Lemma 5.23 *Let M be a λ_f term that represents a De Bruijn term and is in $E \cup F$ normal form. Then,*

$$\varepsilon(M[N],\langle f, g\circ h\rangle) =_{E,F} \varepsilon(M[\varepsilon(N,g)],\langle f,h\rangle)$$

Proof. We show by induction on the structure of M that

$$\begin{array}{l} \varepsilon(Subst[M,N,n,P^n(Id)],P^n(\langle f,g\circ h\rangle)) \\ \qquad =_{E,F} \varepsilon(Subst[M,\varepsilon(N,f),n,P^n(Id)],P^n(\langle f,h\rangle)). \end{array}$$

Since M represents a De Bruijn term and is in $E \cup F$ normal form, we have to consider only the following three cases. (Note that $M \not\equiv i$.)

1. $M \equiv \varepsilon(i,k!)$: First, note that Proposition 5.5 implies that $k! \circ P^n(Id) \xrightarrow[F]{} \!\!\!\!\to k!$ for any k. We distinguish the following three subcases.

   (a) $k < n$: In this case, $Subst$ on both sides has no effect. Then,

$$\begin{array}{llll} LHS & = & \varepsilon(\varepsilon(i,k!),P^n(\langle f,g\circ h\rangle)) & \\ & =_{E,F} & \varepsilon(i,k!), & \text{by } E \text{ rules and Proposition 5.5,} \\ RHS & = & \varepsilon(\varepsilon(i,k!),P^n(\langle f,h\rangle)) & \\ & =_{E,F} & \varepsilon(i,k!), & \text{by } E \text{ rules and Proposition 5.5.} \end{array}$$

The result follows.

(b) $k = n$: In this case $Subst$ on both sides replaces i by the expression being substituted. Then,

$$\begin{aligned}
LHS &= \varepsilon(\varepsilon(N, k!), P^n(\langle f, g \circ h \rangle)) \\
&=_{E,F} \varepsilon(N, k! \circ P^n(\langle f, g \circ h \rangle)) \\
&=_{E,F} \varepsilon(N, g \circ h \circ Fst^n), && \text{by Proposition 5.5 and } F \text{ rules,} \\
RHS &= \varepsilon(\varepsilon(\varepsilon(N, g), k!), P^n(\langle f, h \rangle)) \\
&=_{E,F} \varepsilon(N, g \circ k! \circ P^n(\langle f, h \rangle)) \\
&=_{E,F} \varepsilon(N, g \circ h \circ Fst^n), && \text{by Proposition 5.5 and } F \text{ rules.}
\end{aligned}$$

The result follows.

(c) $k > m$: In this case, $Subst$ has no effect. Then,

$$\begin{aligned}
LHS &= \varepsilon(\varepsilon(i, k!), P^n(\langle f, g \circ h \rangle)) \\
&=_{E,F} \varepsilon(i, Snd \circ Fst^{k-n} \circ \langle f, g \circ h \rangle \circ Fst^n), \\
&\qquad \text{by } E \text{ rules and Proposition 5.5,} \\
&=_{E,F} \varepsilon(i, Snd \circ Fst^{k-n-1} \circ f \circ Fst^n), \\
RHS &= \varepsilon(\varepsilon(i, k!), P^n(\langle f, h \rangle)) \\
&=_{E,F} \varepsilon(i, Snd \circ Fst^{k-n} \circ \langle f, h \rangle \circ Fst^n), \\
&\qquad \text{by } E \text{ rules and Proposition 5.5} \\
&=_{E,F} \varepsilon(i, Snd \circ Fst^{k-n-1} \circ f \circ Fst^n).
\end{aligned}$$

The result follows.

2. $M \equiv \lambda.R$: By Proposition 5.12, R also represents a De Bruijn term. Then, by definition of the substitution operation and E rules, we have

$$\begin{aligned}
LHS &= \lambda.\varepsilon(Subst[R, N, n+1, P^{n+1}(Id)], P^{n+1}(\langle f, g \circ h \rangle)), \text{ and} \\
RHS &= \lambda.\varepsilon(Subst[R, N, n+1, P^{n+1}(Id)], P^{n+1}(\langle f, h \rangle)).
\end{aligned}$$

The result then follows from the induction hypothesis.

3. $M \equiv R\,S$: Follows easily from the definition of the substitution operation and the induction hypothesis.

This completes the proof of the lemma. □

Lemma 5.24 *Let $M \equiv \varepsilon((\lambda.R)\,S, f)$ be a λ_f-term that represents a De Bruijn term. Then, the following diagram commutes:*

$$
\begin{array}{ccc}
M \equiv \varepsilon(\underline{(\lambda.R)\,S}, f) & \xrightarrow[\beta]{\hspace{6em}} & M' \\
\Big\downarrow{\scriptstyle \varepsilon\text{-}ap,\varepsilon\text{-}\lambda} & & \Big\vert \\
(\lambda.\varepsilon(R, P(f)))\,\varepsilon(S, f) & & \Big\vert{\scriptstyle E,F} \\
\Big\downarrow{\scriptstyle E,F} & & \Big\downarrow \\
N \equiv (\lambda.R')\,\varepsilon(S, f) & \dashrightarrow_{\beta} N'' \dashrightarrow\!\!\!\!\rightarrow_{E,F} & N'
\end{array}
$$

where R' is the $E \cup F$ normal form of $\varepsilon(R, P(f))$.

Proof.

$$
\begin{array}{rlll}
M' & = & \varepsilon(\varepsilon(R[A], \langle Id, Id \rangle), f) & \text{by } \beta_f \text{ rule} \\
 & =_{E,F} & \varepsilon(R[A], \langle f, f \rangle) & \\
 & =_{E,F} & \varepsilon(R[A], P(f) \circ \langle Id, f \rangle) & \text{by definition of } P \\
 & =_{E,F} & \varepsilon(\varepsilon(R[A], P(f)) \langle Id, f \rangle) & \\
 & =_{E,F} & \varepsilon(\varepsilon(Subst[R, S, 0, P(f)], P(f)) \langle Id, f \rangle), & \text{by Proposition 5.21} \\
 & =_{E,F} & \varepsilon((\varepsilon(R, P(f))[S]) \langle Id, f \rangle), & \text{by definition of } Subst \\
 & =_{E,F} & \varepsilon(R'[S], \langle Id, \rangle f), & \text{by assumption} \\
 & & & \text{and Proposition 5.22} \\
 & \equiv & M'' &
\end{array}
$$

And

$$N'' = \varepsilon(R'[\varepsilon(S, f)], \langle Id, Id \rangle), \quad \text{by } (\beta_f) \text{ rule}$$

Since M represents a De Bruijn term, Proposition 5.12 implies that $\varepsilon(R, P(f))$ also represents a De Bruijn term. Thus, R' represents a De Bruijn term and is in $E \cup F$ normal form. Applying Proposition 5.23, we get $N'' =_{E,F} M'' =_{E,F} M'$. Since $E \cup F$ rules are confluent, there exists a N' such that $N'' \underset{E,F}{\twoheadrightarrow} N' \underset{E,F}{\twoheadleftarrow} M'$. □

Now, we can give the proof of Theorem 5.17.

Proof of Theorem 5.17: Let $M \equiv C[(\lambda.R)\ S]$ where the displayed β_f redex is the one reduced in $M \longrightarrow M'$. Since $E \cup F$ reductions are confluent, we may decompose the reduction $M \underset{E,F}{\twoheadrightarrow} N$ as follows:

$$M \underset{E,F}{\twoheadrightarrow} C'[(\lambda.R)\ S'] \equiv T \underset{E,F}{\twoheadrightarrow} N$$

where the reduction $M \twoheadrightarrow T$ is totally in $C[]$ and S. Moreover, $C'[]$ is the $E \cup F$ normal form of $C[]$, and S' is the $E \cup F$ normal form of S.

Then, we have two cases. Either T contains an ε term such $(\lambda.R)\ S'$ is a subterm of the ε term, or there is no such ε term in T. In the first case, the result follows easily from Proposition 5.22. So, consider the second case.

Since C' is in $E \cup F$ normal form, T must be of the form

$$T \equiv C''[\varepsilon((\lambda.R)\ S', f)]$$

for some context C'' in $E \cup F$ normal form. Moreover, C'' contains no ε term that includes the expression in the hole.

Thus, by Proposition 5.12, $\varepsilon((\lambda.R)\ S', f)$ represents a De Bruijn term.

The reduction $T \longrightarrow N$ can be decomposed as follows:

$$\begin{array}{lll} T & \underset{E,F}{\twoheadrightarrow} & C''[(\lambda.\varepsilon(R, P(f)))\ \varepsilon(S, f)] \equiv T' \\ & \underset{E,F}{\twoheadrightarrow} & C''[(\lambda.R')\ \varepsilon(S, f)] \\ & \underset{E,F}{\twoheadrightarrow} & N \end{array}$$

where R' is the $E \cup F$ normal form of $\varepsilon(R, P(f))$.

The result then follows from the following diagram:

$$\begin{array}{ccccc} M & & \xrightarrow{\quad \beta_f \quad} & & M' \\ \downarrow{\scriptstyle E,F} & & & & \downarrow{\scriptstyle E,F} \\ T & & \dashrightarrow_{\beta_f} & & \bullet \\ \downarrow{\scriptstyle \varepsilon\text{-}ap,sel} & & & & \downarrow{\scriptstyle E,F} \\ T' & & & \dashrightarrow_{E,F} & \downarrow{\scriptstyle E,F} \\ \downarrow{\scriptstyle E,F} & & & & \downarrow \\ N & \dashrightarrow_{\beta_f} & N'' & \dashrightarrow_{E,F} & N' \end{array}$$

The top part of the diagram comes from repeated application of Proposition 5.19, and the bottom part of the diagram is simply Proposition 5.24. □

We end this section with a remark. We believe that λ_f terms that represent De Bruijn terms and F E and (β_f) form a confluent system. The only case that we didn't consider is the (β_f)-(β_f) case, for which the standard techniques used in proving the confluence of the λ-calculus should suffice.

5.8 Equivalence of λ_f calculus and De Bruijn notation

In this section, we show the relationship between λ_f calculus and De Bruijn notation. The main theorem is the following:

Theorem 5.25 *Let M be a λ_f term that represents a De Bruijn term, and let A be a De Bruijn term.*

1. *Suppose $M \longrightarrow N$. Then, N represents a De Bruijn term. Moreover,*
   (a) $M \underset{E,F}{\longrightarrow} N \Rightarrow M_{DB} \equiv N_{DB}$, *and*
   (b) $M \underset{\beta_f}{\longrightarrow} N \Rightarrow M_{DB} \longrightarrow N_{DB}$.
2. $A \longrightarrow B \Rightarrow A_{\lambda_f} \underset{\beta_f}{\longrightarrow} A' \underset{E,F}{\twoheadrightarrow} B_{\lambda_f}$
3. $M \underset{FE}{\twoheadrightarrow} (M_{DB})_{\lambda_f}$ *and* $A \equiv (A_{\lambda_f})_{DB}$.

Before we give the proof of the theorem, we show how the lifting and substitution operations of De Bruijn's notation are simulated in λ_f calculus.

Recall that the translation of a λ_f term that represents a De Bruijn term into a De Bruijn term is defined as follows: find $E \cup F$ normal form of the term and replace $\varepsilon(i, n!)$ by n. Thus, the only difference between λ_f terms that represent De Bruijn terms and are in $E \cup F$ normal form and their translation into De Bruijn notation is in the notation used for De Bruijn numbers.

Lemma 5.26 *Let M be a term in De Bruijn's notation. Then,*

1. $\varepsilon(M_{\lambda_f}, P^m(Id)) \underset{E,F}{\twoheadrightarrow} (U^0_m(M))_{\lambda_f}$.
2. $\varepsilon(M_{\lambda_f}, P^m(Fst^n)) \underset{E,F}{\twoheadrightarrow} (U^n_m(M))_{\lambda_f}$, *for* $n \geq 1$.

Proof. We prove only the second part; the proof of the first part is similar. The proof is by induction on the structure of M. Let LHS denote the left hand side of the reduction.

1. $M \equiv k$. Then, $M_{\lambda_f} = \varepsilon(i, k!)$. As in the definition of U_m^n, we consider the following two cases.

   (a) $k < m$. Then $U_m^n(k) = k$, and

$$\begin{array}{llll} LHS & \underset{e\text{-}e}{\longrightarrow} & \varepsilon(i, k! \circ P^m(Fst^n)), & \\ & \underset{F}{\twoheadrightarrow} & \varepsilon(i, k!), & \text{by Proposition 5.5(1)}, \\ & = & k_{\lambda_f}. & \end{array}$$

   (b) $k \geq m$. Then $U_m^n(k) = k + n$, and

$$\begin{array}{llll} LHS & \underset{e\text{-}e}{\longrightarrow} & \varepsilon(i, k! \circ P^m(Fst^n)), & \\ & \underset{F}{\twoheadrightarrow} & \varepsilon(i, Snd \circ Fst^{k-m} \circ (Fst^n) \circ Fst^m), & \text{by Proposition 5.5(2)}, \\ & \underset{F}{\twoheadrightarrow} & \varepsilon(i, (k+n)!), & \\ & = & (k+n)_{\lambda_f}. & \end{array}$$

2. $M \equiv N\ A$. Then,

$$\begin{array}{llll} LHS & = & \varepsilon(N_{\lambda_f}\ A_{\lambda_f}, P^m(Fst^n)), & \\ & \underset{e\text{-}ap}{\longrightarrow} & \varepsilon(N_{\lambda_f}, P^m(Fst^n))\, \varepsilon(A_{\lambda_f}, P^m(Fst^n)), & \\ & \underset{E,F}{\twoheadrightarrow} & (U_m^n(N))_{\lambda_f}\ (U_m^n(A))_{\lambda_f}, & \text{by induction hypothesis}, \\ & = & (U_m^n(N\ A))_{\lambda_f}, & \text{by the definition of } U_m^n. \end{array}$$

3. $M \equiv \lambda.N$. Then,

$$\begin{array}{llll} LHS & = & \varepsilon(\lambda.N_{\lambda_f}, P^m(Fst^n)) & \\ & \underset{e\text{-}\lambda}{\longrightarrow} & \lambda.\varepsilon(N_{\lambda_f}, P^{m+1}(Fst^n)), & \\ & \underset{E,F}{\twoheadrightarrow} & \lambda.(U_{m+1}^n(N))_{\lambda_f}, & \text{by induction hypothesis}, \\ & = & (U_m^n(\lambda.N))_{\lambda_f} & \text{by the definition of } U_m^n. \end{array}$$

Thus, the lemma holds. □

Lemma 5.27 *Let M, N be terms in DeBruijn's notation. Then,*

$$\varepsilon(M_{\lambda_f}[N_{\lambda_f}], \langle Id, Id \rangle) \underset{E,F}{\twoheadrightarrow} (M[0 := N])_{\lambda_f}$$

Proof. By induction on the structure of M, we show that

$$\varepsilon(Subst[M_{\lambda_f}, N_{\lambda_f}, m, P^m(Id)], P^m(\langle Id, Id \rangle)) \underset{E,F}{\twoheadrightarrow} (M[m := N])_{\lambda_f}$$

We consider the following there cases. Let LHS denote the left hand side of the above equation.

1. $M \equiv k$. Then, $M_{\lambda_f} = \varepsilon(i, k!)$, and by the definition of $Subst$

$$\begin{array}{rcl} LHS & = & \varepsilon(\varepsilon(Subst[i, N_{\lambda_f}, m, k! \circ P^m(Id)], k!), P^m(\langle Id, Id\rangle)) \\ & \xrightarrow[\varepsilon\text{-}\varepsilon]{} & \varepsilon(Subst[i, N_{\lambda_f}, m, k! \circ P^m(Id)], k! \circ P^m(\langle Id, Id\rangle)) \end{array}$$

By Proposition 5.5(1) and (2), $k! \circ P^m(Id)$ reduces to $m!$ only when $k! = m!$, that is, $k = m$. Thus,

$$Subst[i, N_{\lambda_f}, m, k! \circ P^m(Id)] = \begin{cases} N_{\lambda_f} & \text{if } k = m \\ i & \text{otherwise} \end{cases}$$

Now, we consider the following three cases similar to the three cases in the definition of the substitution operation for DeBruijn's notation.

(a) $k < m$. Then,

$$\begin{array}{rcll} LHS & = & \varepsilon(i, k! \circ P^m(\langle Id, Id\rangle)), & \\ & \xrightarrow[F]{} & \varepsilon(i, k!), & \text{by Proposition 5.5(1),} \\ & = & k_{\lambda_f} = (M[m := N])_{\lambda_f}. & \end{array}$$

(b) $k = m$. We suppose that $k > 0$. The proof for $k = 0$ is similar but uses Proposition 5.26(1).

$$\begin{array}{rcll} LHS & = & \varepsilon(N_{\lambda_f}, k! \circ P^m(\langle Id, Id\rangle)), & \\ & \xrightarrow[F]{} & \varepsilon(N_{\lambda_f}, Snd \circ \langle Id, Id\rangle \circ Fst^m), & \text{by Proposition 5.5(2),} \\ & \xrightarrow[F]{} & \varepsilon(N_{\lambda_f}, Fst^m), & \\ & \xrightarrow[E,F]{} & (U^n_m(N))_{\lambda_f} & \text{by Proposition 5.26(1)} \\ & = & (M[m := N])_{\lambda_f}, & \end{array}$$

(c) $k > m$. In this case,

$$\begin{array}{rcll} LHS & = & \varepsilon(i, k! \circ P^m(\langle Id, Id\rangle)), & \\ & \xrightarrow[F]{} & \varepsilon(i, Snd \circ Fst^{k-m} \circ \langle Id, Id\rangle \circ Fst^m), & \text{by Proposition 5.5(2),} \\ & \xrightarrow[F]{} & \varepsilon(i, (k-1)!), & \\ & = & (k-1)_{\lambda_f} = (M[m := N])_{\lambda_f}. & \end{array}$$

2. $M \equiv R\,A$. By the definition of $Subst$ operation and ε-ap rule,

$$\begin{aligned} LHS &\longrightarrow \varepsilon(Subst[R_{\lambda_f}, N_{\lambda_f}, m, P^m(Id)], P^m(\langle Id, Id\rangle)) \\ &\qquad \varepsilon(Subst[A_{\lambda_f}, N_{\lambda_f}, m, P^m(Id)], P^m(\langle Id, Id\rangle)), \end{aligned}$$

which, by the induction hypothesis,

$$\begin{aligned} &\underset{E,F}{\twoheadrightarrow} (R[m := N])_{\lambda_f} (A[m := N])_{\lambda_f} \\ &= (M[m := N])_{\lambda_f}. \end{aligned}$$

3. $M \equiv \lambda.R$. In this case, we use ε-λ rule. Thus,

$$LHS \longrightarrow \lambda.\varepsilon(Subst[R_{\lambda_f}, N_{\lambda_f}, m+1, P^{m+1}(Id)], P^{m+1}(\langle Id, Id\rangle)),$$

which, by the induction hypothesis,

$$\begin{aligned} &\underset{F,E}{\twoheadrightarrow} \lambda.(R[m+1 := N])_{\lambda_f} \\ &= (M[m := N])_{\lambda_f} \end{aligned}$$

This completes the proof. □

Since λ_f terms that represent De Bruijn terms and are in $E \cup F$ normal form are essentially identical to De Bruijn terms (modulo encoding of numbers), the following proposition is actually a restatement of Proposition 5.27.

Lemma 5.28 *Let M, N be λ_f terms that represent De Bruijn terms, and suppose M and N are in $E \cup F$ normal form. Then, $\varepsilon(M[N], \langle Id, Id\rangle)$ represents a De Bruijn term and*

$$\varepsilon(M[N], \langle Id, Id\rangle)_{DB} = M_{DB}[0 := N_{DB}]$$

Proof. Similar to the proof of Proposition 5.27, we show that

$$\varepsilon(M[N], \langle Id, Id\rangle) \underset{E,F}{\twoheadrightarrow} (M_{DB}[0 := N_{DB}])_{\lambda_f}$$

This immediately shows that the term $\varepsilon(M[N], \langle Id, Id\rangle)$ represents a De Bruijn term. To prove the second part, we note that for λ_f terms in $E \cup F$ normal form $()_{DB}$ is the inverse of $()_{\lambda_f}$.

□ *Proof of Theorem 5.25:*

Part 1. We show that N represents a De Bruijn term separately for the two cases.

(a) The result that N represents a De Bruijn term follows from Definition 5.10 and the confluence of $E \cup F$ rules. The result that $M_{DB} = N_{DB}$ follows from the definition of the translation and the confluence of $E \cup F$ rules.

(b) First, suppose M is in $E \cup F$ normal form. Let $M \equiv C[(\lambda.R)\ A]$ where the displayed occurrence of β_f redex is the one contracted in the reduction of M to N. Then,

$$N \equiv C[\varepsilon(R[A], \langle Id, Id \rangle)]$$

Since M is in $E \cup F$ normal form, $C[]$, R, and A are also in $E \cup F$ normal form. Since M represents a De Bruijn term and $C[]$ contains no ε that includes the term in the hole as a subterm, Proposition 5.12 implies that both R and A represent De Bruijn terms. By Proposition 5.28, $\varepsilon(R[A], \langle Id, Id \rangle)$ represents a De bruijn term and

$$\varepsilon(R[A], \langle Id, Id \rangle)_{DB} = R_{DB}[0 := A_{DB}]$$

Thus, N represents a De Bruijn term, which proves the first part. Also,

$$N_{DB} = C_{DB}[R_{DB}[0 := A_{DB}]]$$

On the other hand,

$$\begin{array}{rcl} (M)_{DB} & = & C_{DB}[(\lambda.R_{DB})\ A_{DB}], \\ & \xrightarrow[\beta]{} & C_{DB}[R_{DB}[0 := A_{DB}]]. \end{array}$$

Thus, $M_{DB} \xrightarrow[\beta]{} N_{DB}$, which proves the second part.

Now, suppose M is not in $E \cup F$ normal form. Let M' be the $E \cup F$ normal form of M. Combining the diagram from Proposition 5.17 and the argument given above, we have the following diagram.

$$\begin{array}{ccccc} M & & \xrightarrow[\beta_f]{\hspace{6em}} & & N \\ {\scriptstyle E,F}\downarrow\!\!\downarrow & & & & \downarrow\!\!\downarrow{\scriptstyle E,F} \\ M' & \xrightarrow[\beta_f]{} & M'' & \xrightarrow[E,F]{} & N' \\ {\scriptstyle ()_{DB}}| & & {\scriptstyle ()_{DB}}| & & {\scriptstyle ()_{DB}}| \\ M_{DB} & \longrightarrow & (M'')_{DB} & \equiv & N_{DB} \end{array}$$

Both results follow from this diagram.

Part 2. Suppose A is of the form $C[(\lambda.R)\ A]$ where the displayed occurrence of β redex is the one reduced in $A \longrightarrow B$. Then, $B \equiv C[R[0 := A]]$. On the other hand,

$$\begin{array}{rcl} A_{\lambda_f} & = & C_{\lambda_f}[(\lambda.R_{\lambda_f})\ A_{\lambda_f}], \\ & \xrightarrow[\beta_f]{} & C_{\lambda_f}[\varepsilon(R_{\lambda_f}[A_{\lambda_f}], \langle Id, Id \rangle)] \end{array}$$

Let A' be this last term. By Proposition 5.27,

$$\varepsilon(Subst[R_{\lambda_f}[A_{\lambda_f}], 0, Id], \langle Id, Id\rangle) \underset{E,F}{\twoheadrightarrow} (R[0 := A])_{\lambda_f}.$$

Since C_{λ_f} is in $E \cup F$ normal form, $A' \underset{E,F}{\twoheadrightarrow} B_{\lambda_f}$. This completes the proof.

Part 3. It follows easily from the definitions of $(\ )_{DB}$ and $(\ )_{\lambda_f}$. □

5.9 Reduction strategies

There are a number of ways to apply the reduction rules of λ_f calculus that are normalizing. We present two of them.

1. Apply rules F, E, and β_f in the leftmost-outermost way.

2. Find the normal form with respect to rules β, β', and ε-ε by applying these rules in the leftmost-outermost way. Then apply rules F and E in the leftmost-outermost way. To perform substitution, it will be necessary to use F reduction rules; however, these rules are not applied to the term.

Both these strategies are normalizing for λ_f calculus. More importantly, if a λ_f term represents a De Bruijn term, then its reduction according to either of the strategy terminates iff its translation into a De Bruijn term has a normal form. The second strategy of reducing terms substantiates our claim that all modification to De Bruijn numbers can be postponed until the expression is in β normal form.

Proposition 5.29 *Let A be a λ-term in De Bruijn notation. The reduction of A_{λ_f} using either of the two strategies given above terminates iff A has a normal form.*

Proof. We show the proposition only for the second strategy, which is the more interesting one of the two. We make a couple of observations. First, consider a β_f or β'_f step in the reduction of A_{λ_f} using the second strategy. Suppose the step is $M \longrightarrow N$, and suppose R is the redex reduced at this step. Then, the corresponding step $M_{DB} \longrightarrow N_{DB}$ reduces the leftmost redex. Suppose not. Consider the leftmost redex in M_{DB}. Corresponding to this redex, there must be an application and a λ-abstraction in M, both of which are to the left of R. Moreover, either they form a β_f redex or they are separated by some ε's. In either case R is not the redex that will be reduced at this step. Second, if a λ_f term is in normal form relative to (ε-ε), (β_f) and

(β'_f) rules, then its translation into a De Bruijn term is also in normal form. The argument is similar to the one given above.

Now, suppose A has a normal form, but the reduction of A_{slf} using the strategy doesn't terminate. Then, the reduction must contain infinite number of β_f steps, since $E \cup F$ rules are noetherian. Then, the corresponding reduction starting from A is leftmost and infinite, which is impossible.

Now, suppose A has no normal form, but the reduction of A_{slf} using the strategy terminates. Consider the reduction that corresponds to the first part of the strategy, *i.e.*, finding normal form relative to (ε-ε), (β_f) and (β'_f) rules. Then the corresponding reduction starting from A is leftmost and ends in a term in normal form, which is impossible. □

5.10 Summary

In this chapter, we introduced a formal system, called λ_f calculus, which is based on De Bruijn's notation. The system will be used as an intermediate step in proving the correctness of the interpreter; however it may be of independent interest as it offers certain advantages over either De Bruijn notation or the λ-calculus for designing graph reduction interpreters for the λ-calculus.

Chapter 6

Proofs of Correctness and Optimality

6.1 Introduction

In this chapter, we prove that the interpreter is a correct implementation of the λ-calculus and that it realizes Lévy's specification of optimal reductions in the λ-calculus.

To prove the correctness, we have to show the following. Given a λ-expression that has normal form, the interpreter reduces the initial graph representation of the expression to a graph that translates to the normal form of expression. On the other hand, if a λ-expression has no normal form, then the reduction of the initial graph representation of the expression doesn't terminate. The usual way to prove such results is to establish a step-wise correspondence. In our case, we show that the diagrams given below commute. In these diagrams, G and G' are λ_{fc} graphs; M and M' are λ-terms in De Bruijn notation; and $\xrightarrow[Tr]{}$ denotes translation of λ_{fc} graphs to λ-terms.

$$\begin{array}{ccc} G & \xrightarrow[\beta_{fc}]{} & G' \\ Tr \big\downarrow & & Tr \big\downarrow \\ M & \underset{\beta}{\dashrightarrow\!\!\!\!\rightarrow} & M' \end{array} \qquad\qquad \begin{array}{ccc} G & \xrightarrow[E_g,F_g]{} & G' \\ Tr \big\downarrow & & Tr \big\downarrow \\ M & \equiv & M \end{array}$$

The result follows from these diagrams and a couple of simple observations. First, the initial graph representation of a λ-term translates to the λ-term; this forms the base case for composing these diagrams. Second, the computation rule used by the interpreter is a variation of the leftmost rule in the λ-calculus and guarantees that the interpreter terminates iff the initial λ-expression has normal form.

We will prove the correctness of the interpreter in two parts. First, we introduce a term system, called λ_{fc} calculus, which is quite close to the graph reduction system underlying the interpreter in the following sense. The terms of the calculus are finite trees obtained by unraveling λ_{fc} graphs and throwing away one side of each conditional node. The reduction rules of the calculus are term versions of the graph reduction rules with a few exceptions discussed later. The calculus is an extension of the λ_f calculus introduced in Chapter 5, and we show the correspondence between the calculii. In the second part of the proof, we show the correspondence between λ_{fc} calculus and the interpreter.

The reason for adopting this two-part strategy is that it seems too difficult to give a direct proof because of two things: cyclic graphs, and the way graphs are translated into λ-expressions.[1] The combination of the two is hard to handle in a straightforward way. Suppose we want to prove that the diagrams given above hold, and suppose G is cyclic. It is easier to think in terms of trees. So, consider the tree obtained by unraveling G. The tree, of course, will be infinite but that is not relevant to the discussion. The reduction of G to G' corresponds to simultaneous reduction of a set of redexes in the tree. Since G is cyclic, redexes in this set are not necessarily disjoint and may be nested in a complex way. For example, in the case of β_{fc} reduction, both the function part and the argument part of a redex in this set may contain other redexes in the set. Now, consider two nested redexes in this set of redexes. To relate the translations of the outer redex and its reduct, we must first relate the translations of the inner redex and its reduct. On the other hand, relating the subexpressions represented by the inner redex and its reduct requires that we know the functions on control environments used to translate them. But, the function used to translate the reduct of the inner redex cannot be determined without taking into account the reduction of the outer redex (see the rules (β_{fc}) and (λ-sp)). This need to propagate the effect of a reduction in both bottom-up and top-down manner makes it difficult to give a nice inductive argument.

The proof would be simpler if either graphs weren't cyclic or the translation was as simple as unraveling graphs. This observation suggests that we should proceed as follows. First, we should unravel graphs into trees and take care of sharing and cycles present in graphs. Then, we translate trees into λ-expressions and take care of the interpretation of conditionals, which is easier to handle for trees (or even acyclic graphs) than for general λ_{fc} graphs. The problem is that unraveling of cyclic graphs results into infinite trees, and we certainly don't wish to work

[1] Besides, it is no fun.

with infinite trees.

Notice, however, the following. First, the translation of a conditional node in a graph replaces the conditional node by one of its sides; the side that is chosen is based on the structure of the path followed by the translation procedure to reach the conditional node. In a tree, there is only one path to each conditional node, so one side of each conditional node is always useless and will simply be thrown away by the translation procedure. Second, the λ_{fc} graphs that are of interest are the ones that can be translated to λ-expressions. If we unravel such a graph and throw away the useless side of each conditional node, then the resulting tree must be finite regardless of whether the graph is cyclic or not.

So, we can convert λ_{fc} graphs that are of interest to us into finite trees by unraveling them and throwing away the useless sides of conditional nodes. Finding the useless side of a conditional node, however, is nothing but interpreting the conditional, something we don't want to do while converting graphs into trees.

Thus, we proceed as follows. We consider all finite trees (or terms) that can obtained from λ_{fc} graphs in the following manner. Given a λ_{fc} graph, unravel the graph and replace one side of each conditional node, not necessarily the useless side, by the symbol $\Box$. Terms obtained in this way are terms of the λ_{fc} calculus. Not all of these terms represent De Bruijn terms, and we identify a subset of λ_{fc} terms that do. The two main properties of this subset of terms are as follows. First, the $\Box$ side of each conditional in a term in this subset is indeed the useless side. Second, the subset is closed under reduction by the rules of the λ_{fc} calculus.

Now, given a λ_{fc} graph, there is a set of λ_{fc} terms that can be obtained from the graph in the way described above. Each term in this set corresponds to one possible way of interpreting conditional nodes in the graph to get a finite term. If a λ_{fc} graph can be translated into a De Bruijn term, then the set of λ_{fc} terms associated with the graph must contain an unique λ_{fc} term that represents a De Bruijn term. We simply take this (with one other condition discussed later) to be the definition of graphs that represent De Bruijn terms. Furthermore, the translation of a such a graph into λ_{fc} calculus is simply the unique λ_{fc} term that represent a De Bruijn term. Notice also that if a graph cannot be translated into a De Bruijn term, then either the set of terms associated with the graph is empty,*i.e.*, there is no way to interpret conditionals to get a finite term, or the set doesn't contain a λ_{fc} term that can be translated into a De Bruijn term.

To show the correspondence between the interpreter and the λ_{fc} calculus, we have to show

the following. Suppose G is a λ_{fc} graph that represents a De Bruijn term. If $G \longrightarrow G'$, then the translation of G reduces to the translation of G'. Given that the set of λ_{fc} terms that represent De Bruijn terms is closed under reduction, the only thing we have to show is that the translation of G reduces to a λ_{fc} term that belongs to the set of terms associated with G', which is much easier to prove than to show that the translation of G reduces to the translation of G'.

Now, we come to the proof of optimality. We show that the interpreter correctly implements Lévy's theoretical specification of optimal reductions in the λ-calculus. More precisely, suppose M is a λ-term that has a normal form and M_L is a labelled term obtained by labelling each subexpression of M by elementary labels. Now, suppose we reduce the initial graph representation of M using the interpreter. Then, we show that each β_{fc} step in the reduction performed by the interpreter corresponds to a step in the leftmost complete reduction beginning with M_L. In other words, the reduction performed by the interpreter corresponds to the leftmost complete reduction with M_L, and the number of β_{fc} redexes reduced by the interpreter equals the length of the leftmost complete reduction.

To show the result, we have to establish a correspondence between the structure of the graph at a step with the labels of subexpressions in the corresponding labeled expression. The correspondence is somewhat involved for the following reasons. First, to get an inductive argument, we must establish the correspondence not only for redexes but also for other types of subexpressions. Second, labels in the labelled calculus change only during β-reduction whereas the interpreter also uses other rules that change the structure of a graph. Moreover, rules (λ-sp) and (Ap-c) change the structure of graph anticipating that certain β_{fc} redexes will be performed at subsequent step.

To establish the correspondence, we define a relation called, *l-shared* on the subexpressions of a labelled expression. Redexes that have the same degree and must be done in one step are always l-shared; indeed, this will be the base case of the inductive definition. Furthermore, the relation is such that only l-shared subexpressions contribute to the creation of redexes that have the same degree.

The rest of the chapter is organized as follows. In Section 6.2, we describe the λ_{fc} calculus and show its correspondence with the λ_f calculus. In Section 6.3, we describe the general notion of *approximating* graphs by terms. In Section 6.4, we give proof of the correctness of the interpreter, and in Section 6.5, we give proof the optimality.

6.2 λ_{fc} calculus

In this section, we extend the λ_f-calculus by adding conditional expressions. The calculus captures most features, except the proper treatment of the β-rule, of our interpreter. It will be used in proving the correctness of our interpreter.

6.2.1 Functions on control environments

The structure of control environments is the same as in Section 4.2 Functions on control environments and reduction rules are the same as in Section 4.4.1.

Theorem 6.1 *The system $(\mathbf{F}_c, F_c)$ is noetherian and confluent.*

Proof. The proof is similar to the proof of Theorem 5.4 in the last chapter. Note that the system $(\mathbf{F}, F)$ is a subsystem of $(\mathbf{F}_c, F_c)$.

Noetherian: We define a polynomial interpretation $(-)_{\tau'}$ that maps terms in $\mathbf{F}_c$ to natural numbers as follows.

$$Empty_{\tau'}, L_{\tau'}, R_{\tau'}, Pop_{\tau'}, Env_{\tau'} \quad = \quad 2$$

For all other types of terms $(-)_{\tau'}$ is defined as $(-)_{\tau}$.

We only have to verify that $l_\tau > r_\tau$ holds for each of the new rules, *i.e.*, rules that are not in F. The result is trivial to verify as the right hand side of each of the new rules is smaller in size than

Local confluence: We have to check local confluence only for each pair of interfering rules such that either both rules are new rules, or one of the rules is a new rule and the other rule is a F rule. The new rules don't interfere with each other. Among the F rules, only (Ass) interferes with the new rules, and it interferes with each one of them. In each case, it is easy to check for local confluence. Indeed, the rules (PopL'), (PopR') and (Env) were introduced to make the system confluent. We show local confluence for two cases; other cases are similar.

1. (Ass) and (PopL):

$$\begin{array}{ccc} (Pop \circ L) \circ f & \xrightarrow{Ass} & Pop \circ (L \circ f) \\ \Big\downarrow{\scriptstyle PopL} & & \Big\downarrow{\scriptstyle PopL'} \\ Id \circ f & \xrightarrow[IdL]{} \!\!\!\!\rightarrow & f \end{array}$$

2. (Ass) and (PopL'):

$$\begin{array}{ccc}
(Pop \circ (L \circ f)) \circ g & \xrightarrow[Ass]{} & Pop \circ ((L \circ f) \circ g) \\
\Big\downarrow {\scriptstyle PopL'} & & \Big\downarrow {\scriptstyle Ass, PopL'} \\
f \circ g & \equiv & f \circ g
\end{array}$$

Confluence follows from Newman's Lemma. □

We will use the notation introduced before Proposition 5.5 in the last chapter. Also, Proposition 5.5 remains valid in the extended system $(\mathbf{F}_c, F_c)$. Any reference to that proposition in subsequent section means the corresponding proposition for the system $(\mathbf{F}_c, F_c)$.

6.2.2 λ_{fc} terms and E_c reduction rules

Definition 6.2 The syntax of terms of the λ_{fc} calculus is as follows:

$$\begin{array}{rcl}
Term & ::= & i \mid \lambda.Term \mid Term\ Term \mid \varepsilon(Term, f) \\
 & & \mid cond(f, Term, \Box) \mid cond(f, \Box, Term)
\end{array}$$

where $f \in \mathbf{F}_c$. The set of terms will be denoted by Λ_{fc}. Variables $M, N, \ldots$ range over Λ_{fc}.

Following the nomenclature used in describing the interpreter, the second component of a conditional expression will be called the left hand side of the conditional and the third component will be called right hand side.

Definition 6.3 E_c is the following set of reduction rules:

(ε-ε): $\varepsilon(\varepsilon(M, f), g) \longrightarrow \varepsilon(M, f \circ g)$

(ε-ap): $\varepsilon(M\ N, f) \longrightarrow \varepsilon(M, f)\, \varepsilon(N, f)$

(ε-λ): $\varepsilon(\lambda.M, f) \longrightarrow \lambda.\varepsilon(M, P(f))$

(ε-cL): $\varepsilon(cond(g, M, \Box), f) \longrightarrow cond(g \circ f, \varepsilon(M, f), \Box)$

(ε-cR): $\varepsilon(cond(g, \Box, M), f) \longrightarrow cond(g \circ f, \Box, \varepsilon(M, f))$

(Ap-cL): $cond(f, M, \Box)\, N \longrightarrow cond(f, M\ N, \Box)$

(Ap-cR): $cond(f, \Box, M)\, N \longrightarrow cond(f, \Box, M\ N)$

(C-L): $cond(L, M, \Box) \longrightarrow M$

(C-L'): $cond(L \circ f, M, \Box) \longrightarrow M$

(C-R): $cond(R, \Box, M) \longrightarrow M$

(C-R'): $cond(R \circ f, \Box, M) \longrightarrow M$

A few remarks about the rules. The first three rules correspond to the similarly named rules in the interpreter. Moreover, they are the same as in the λ_f calculus. The rules (ε-cL) and (ε-cR) correspond to the rule (ε-c) of the interpreter; they push ε past a conditional. Notice that these rules don't push ε to the $\Box$ side. Similarly, the rules (AP-cL) and (AP-cR) correspond to the rule (Ap-c) of the interpreter. The last four rules correspond to the similarly named rules in the interpreter. In these rules, notice the correspondence of function part of the condition and the side on which $\Box$ occurs. In particular, there are no rules to remove conditionals in cases like $cond(L, \Box, M)$ and $cond(R, M, \Box)$.

Theorem 6.4 *The system $(\Lambda_{fc}, E_c \cup F_c)$ is noetherian and confluent.*

Proof. The proof is similar to the proof of Theorem 5.8. Note that the system $(\Lambda_f, E \cup F)$ of the last chapter is a subsystem of $(\Lambda_{fc}, E_c \cup F_c)$.

Noetherian: We extend the polynomial interpretation $(-)_{\tau'}$ used in the Theorem 6.1 to λ_{fc} terms as follows. For ε, application and λ-abstraction terms, $(-)_{\tau'}$ is identical to $(-)_{\tau}$. For conditional terms,

$$\begin{aligned} cond(f, M, \Box)_{\tau'} &= M_{\tau'} + f_{\tau'} + 1, \quad \text{and} \\ cond(f, \Box, M)_{\tau'} &= M_{\tau'} + f_{\tau'} + 1 \end{aligned}$$

We have to show that $l_\tau > r_\tau$ only for the new rules, *i.e.*, E_c rules that are not in E.

1. (ε-cL) and (ε-cR): The proof for both rules is identical. We only show it for (ε-cL).

$$\begin{aligned} l_{\tau'} &= (\varepsilon(cond(g, M, \Box), f))_{\tau'} \\ &= (M_{\tau'} + g_{\tau'} + 1)^2 f_{\tau'} \\ r_{\tau'} &= (cond(g \circ f, \varepsilon(M, f), \Box))_{\tau'} \\ &= g_{\tau'} f_{\tau'} + (M_{\tau'})^2 f_{\tau'} + 1 \end{aligned}$$

   The result follows from simple algebraic manipulations.

2. (Ap-cL) and (Ap-cR): The proof for both rules is identical. We only show it for (Ap-cL).

$$\begin{aligned} l_{\tau'} &= ((cond(f, M, \Box)\, N)_{\tau'} \\ &= 2(M_{\tau'} + f_{\tau'} + 1) + N_{\tau'} \\ r_{\tau'} &= (cond(f, M\, N, \Box))_{\tau'} \\ &= f_{\tau'} + 2M_{\tau'} + N_{\tau'} + 1 \end{aligned}$$

   The result follows.

3. (C-*) family of rules: For each of these rules, the result is trivial to verify as the right hand side is smaller in size than the left hand side.

Thus, the system is noetherian.

Local confluence: We have to check local confluence only for each pair of interfering rules such that either both rules are in E_c and not in E, or one of the rules is in E_c but not in E and the other rule is in E. We have the following cases.

1. (ε-ε) interferes with both (ε-cL) and (ε-cR). We show the result only for the pair (ε-ε) and (ε-cL).

$$\begin{array}{ccc} \varepsilon(\varepsilon(cond(f,M,\Box),g),h) & \xrightarrow[\varepsilon\text{-}\varepsilon]{} & \varepsilon(cond(f,M,\Box),g\circ h) \\ \downarrow{\scriptstyle \varepsilon\text{-}cL} & & \downarrow{\scriptstyle \varepsilon\text{-}\varepsilon,Ass} \\ \varepsilon(cond(f\circ g,\varepsilon(M,g),\Box),h) & \xrightarrow[\varepsilon\text{-}cL,\varepsilon\text{-}\varepsilon,\ Ass]{} & cond(f\circ g\circ h,\varepsilon(M,g\circ h),\Box) \end{array}$$

2. (ε-ap) interferes with both (Ap-cL) and (Ap-cR). We show the result only for the pair (ε-ap) and (Ap-cL).

$$\begin{array}{ccc} \varepsilon(cond(f,M,\Box)\,N,g) & \xrightarrow[\varepsilon\text{-}ap]{} & \varepsilon(cond(f,M,\Box),g)\,\varepsilon(N,g) \\ \downarrow{\scriptstyle Ap\text{-}cL} & & \downarrow{\scriptstyle \varepsilon\text{-}cL,Ap\text{-}cL} \\ \varepsilon(cond(f,M\,N,\Box),g) & \xrightarrow[\varepsilon\text{-}cL,\varepsilon\text{-}ap]{} & cond(f\circ g,\varepsilon(M,g)\,\varepsilon(N,g),\Box) \end{array}$$

3. Each of (ε-cL), (ε-cR), (Ap-cL) and (Ap-cR) interferes with each rule in (C-*) family. We show the result only for the pair (ε-cL) and (C-L).

$$\begin{array}{ccc} \varepsilon(cond(L,M,\Box),f) & \xrightarrow[\varepsilon\text{-}cL]{} & cond(L\circ f,\varepsilon(M,f),\Box) \\ \downarrow{\scriptstyle C\text{-}L} & & \downarrow{\scriptstyle C\text{-}L'} \\ \varepsilon(M,f) & \equiv & \varepsilon(M,f) \end{array}$$

Thus, the system is locally confluent. As usual, confluence follows from Newman's Lemma. □

6.2.3 Translation of λ_{fc} terms into λ_f terms

Since a λ_f term is also a λ_{fc} term, we have to define only how to translate λ_{fc} terms into λ_f terms. Basically, the translation is defined as finding the $E_c \cup F_c$ normal form. However, normal forms of λ_{fc} terms relative to these rules are not necessarily free of conditionals. Furthermore, the λ_{fc} terms of interest are the ones that can be translated to terms in De Bruijn notation. Thus, we define a subset of λ_{fc} terms and define translation for only these terms.

Definition 6.5 A λ_{fc} term *represents a De Bruijn term* iff the $E_c \cup F_c$ normal form of the term is a λ_f term that represents a De Bruijn term.

For any term λ_{fc} term M that represents a De Bruijn term, its translation into a De Bruijn term is denoted by M_{DB} and is defined in the obvious way—find the $E_c \cup F_c$ normal form and translate the λ_f term so obtained into a De Bruijn term (see Section 5.3).

The following proposition shows that if a λ_{fc} term represents a De Bruijn term, then certain types of subexpressions of the term also represent De Bruijn terms. It is similar to the Proposition 5.12 in the last chapter.

Proposition 6.6 *Let $C[M]$ be a λ_{fc} term that represents a De Bruijn term. Suppose the context C contains no ε-term of which M is a subterm, and furthermore, suppose that $M \not\equiv i$. Then, M also represents a De Bruijn term.*

Proof. Similar to the proof of Proposition 5.12. □

The following propositions show some properties of λ_{fc} terms that represent De Bruijn terms; they will be useful in relating the graph reduction system underlying the interpreter and the λ_{fc}-calculus. The first proposition implies that the □ side of a conditional inside λ_{fc} terms representing De Bruijn terms is the useless side.

Proposition 6.7 *Let M be a λ_{fc} term that represent De Bruijn term. First, each conditionals inside M must be eliminated in the reduction to $E_c \cup F_c$ normal form by the application of a rule in the (C-*) family of rules. Second, if the function part of a conditional inside M is L or $L \circ f$, then □ must be on the right hand side of the conditional. Similarly, if the function part is R or $R \circ f$, then □ must be on the left hand side.*

Proof. The proof of the first part is as follows. Note that M represents a De bruijn term. Thus, its $E_c \cup F_c$ normal form is a λ_f term, which doesn't contain any conditionals. Thus,

each conditional in M must be eliminated during the reduction to $E_c \cup F_c$ normal form. By inspection of $E_c \cup F_c$ rules, the only rules that eliminate conditionals are the rules in (C-*) family.

For the second part, we show the result for the case case when the function part is L; proof for the other cases is similar. By the first part, each conditional in M must be eliminated. Now, if the function part of a conditional is L, then the first symbol of the function part remains L until the conditional is eliminated. The only way to eliminate a conditional such that the first symbol of its function part is L is to apply either the (C-L) or (C-L') rule. Both of these rules require that $\Box$ be on the right hand side of the conditional. $\Box$

Proposition 6.8 *Suppose M and M' are λ_{fc} terms such that $M \equiv C[cond(f, N, \Box)]$ and $M' \equiv C[cond(f, \Box, N')]$ where C is a context. Then, both of them can't be λ_{fc} terms that represent De Bruijn terms.*

Proof. By contradiction. Suppose both M and M' represent De Bruijn terms. Consider the reductions of both M and M' to $E_c \cup F_c$ normal forms. Since $E_c \cup F_c$ rules are confluent, we may first reduce $C[]$ to normal form. Then,

$$M \underset{E_c,F_c}{\twoheadrightarrow} C'[cond(f, N, \Box)] \equiv T$$
$$M' \underset{E_c,F_c}{\twoheadrightarrow} C'[cond(f, \Box, N')] \equiv T'$$

$C[]$ reduces to $C'[]$ in both cases follows basically from the confluence of $E_c \cup F_c$ rules; holes play no important role.

Now, we can distinguish two cases. Either $C[]$ contains an ε term that encloses the hole, or it contains no such term. We consider the first case, which is more general. Then,

$$\begin{aligned} T &\equiv C''[\varepsilon(cond(f, N, \Box), g)] \\ &\underset{E_c,F_c}{\twoheadrightarrow} C''[cond(f \circ g, \varepsilon(N, g), \Box)], \text{ and} \\ T &\equiv C''[\varepsilon(cond(f, \Box, N'), g)] \\ &\underset{E_c,F_c}{\twoheadrightarrow} C''[cond(f \circ g, \Box, \varepsilon(N', g))]. \end{aligned}$$

Now, consider the term $cond(f \circ g, \varepsilon(N, g), \Box)]$. Since M represents a De Bruijn term and C'' contains no ε term that encloses the hole, Proposition 6.6 implies that this term also represents a De Bruijn term. By Proposition 6.7, the conditional must be eliminated by the application of a rule in the (C-*) family. The only rules that can eliminate the conditionals in this case are

(C-L) and (C-L'). For these rules to be applicable, it must be the case that $f \circ g$ reduces to either L or $L \circ h$ for some h.

Similarly, the term $cond(f \circ g, \Box, \varepsilon(N', g))$ represents a De Bruijn term and, for that to happen, $f \circ g$ must reduce to R or $R \circ h'$ for some h'.

The requirements placed on $f \circ g$ in the two cases are contradictory, since F_c rules are confluent. Thus, both M and M' cannot represent De Bruijn terms. $\Box$

6.2.4 Traces for $E_c \cup F_c$ reduction

Traces of application, λ-abstractions, and i's relative to a given $E_c \cup F_c$ reduction are defined in the same way as in the λ_f calculus with the following addition for carrying underlining in (Ap-cL) and (Ap-cR) rules.

$$\text{(Ap-cL):}\quad \underline{(}cond(f, M, \Box)\, N) \longrightarrow cond(f, \underline{(}M\, N), \Box)$$

$$\text{(Ap-cR):}\quad \underline{(}cond(f, \Box, M)\, N) \longrightarrow cond(f, \Box, \underline{(}M\, N))$$

Proposition 6.9 *Let M be a λ_{fc} term, and let N be its $E_c \cup F_c$ normal form. Then, any application, λ-abstraction, and i in M has exactly one trace in N and the trace is independent of the reduction from M to N.*

Proof. The uniqueness is proved by inspection of the rules and the definition of traces. To show that trace of a subexpression is independent of the particular choice or reduction to normal form, we can show that confluence holds even after underlining. The proof is similar to the proof of confluence in Theorem 6.4 and is omitted. $\Box$

6.2.5 Free and bound occurrences

For the λ_{fc} calculus, the notions of free and bound occurrences of i in a term are defined as in the λ_f calculus except that now we use $E_c \cup F_c$ rules to find normal form. As pointed out in the last section, the normal form of λ_{fc} terms relative to these rules are not necessarily free of conditionals; however, conditionals don't play any role in defining free and bound occurrences of i.

Proposition 6.10 *A bound occurrence of i in a λ_{fc} term is bound by exactly one λ-abstraction.*

Proof. Similar to the proof of Proposition 5.15.

6.2.6 β reduction

In this section, we introduce a rule that corresponds to the β_{fc} rule of the interpreter. The rule is different from the β_f rule of the λ_f calculus.

$$(\beta_{fc}): \quad (\lambda.M)\,N \longrightarrow \varepsilon(M[\varepsilon(N, Env)], \langle Id, Empty\rangle)$$

Notice that the expression that is substituted is $\varepsilon(N, Env)$ and not simply N. More explicitly,

$$M[N] \;=\; Subst[M, N, 0, Id]$$

where

$$\begin{array}{lcl}
Subst[i, N, n, f] & = & \begin{cases} N & \text{if } f \underset{F_c}{\longrightarrow} Snd \circ Fst^n \\ i & \text{otherwise} \end{cases} \\
Subst[(P\,Q), N, n, f] & = & Subst[P, N, n, f]\; Subst[Q, N, n, f] \\
Subst[\lambda.P, N, n, f] & = & \lambda.(Subst[P, N, n+1, \langle f \circ Fst, Snd\rangle]) \\
Subst[\varepsilon(P, g), N, n, f] & = & \varepsilon(Subst[P, N, n, g \circ f], f) \\
Subst[cond(g, P, \Box), N, n, f] & = & cond(g, Subst[P, N, n, f], \Box) \\
Subst[cond(g, \Box, P), N, n, f] & = & cond(g, \Box, Subst[P, N, n, f])
\end{array}$$

The first four clauses in this definition are the same as in the definition of $Subst$ in the λ_f calculus.

We also introduce rules that correspond to the rule for λ-splitting in the interpreter. These are the rules that introduce conditionals.

$$(\lambda\text{-spL}): \quad (\lambda.M) \longrightarrow \lambda.(\varepsilon(M[cond(Id, \varepsilon(i, Pop), \Box)], \langle Fst, L \circ Snd\rangle)$$

$$(\lambda\text{-spR}): \quad (\lambda.M) \longrightarrow \lambda.(\varepsilon(M[cond(Id, \Box, \varepsilon(i, Pop))], \langle Fst, R \circ Snd\rangle)$$

6.2.7 Traces for β_{fc} and λ-sp

Traces of application, λ-abstraction, and i's for a given β_{fc} reduction are defined as in the λ_f calculus. Traces in the case of λ-spL and λ-spR are defined as follows. Trace of the λ-abstraction on the left hand side of the rule is the λ-abstraction on the right hand side of the rule. Trace of any i that is substituted by a conditional is the i occurring inside the conditional.

6.2.8 Correspondence between λ_{fc} calculus and De Bruijn Notation

In this section, we show that reductions in λ_{fc} calculus correctly simulates reductions in De Bruijn notation. The main theorem is the following.

Theorem 6.11 *Let M be a λ_{fc} term that represents a De Bruijn term. Suppose $M \longrightarrow N$. Then, N represents a De Bruijn term, and moreover,*

1. $M \underset{\neg\beta_{fc}}{\longrightarrow} N \Rightarrow M_{DB} = N_{DB}$,
2. $M \underset{\beta_{fc}}{\longrightarrow} N \Rightarrow M_{DB} \underset{\beta}{\longrightarrow} N_{DB}$.

We prove the theorem by showing the correspondence between λ_{fc} calculus and λ_f calculus. Proposition 6.12 shows that β_{fc} reduction commutes with $E_c \cup F_c$ reduction for λ_{fc} terms that represent De Bruijn terms. Proposition 6.19 shows the correspondence between β_{fc} reduction and β_f reduction for λ_{fc} terms that represent De bruijn terms and are in $E_c \cup F_c$normal form, *i.e.*, λ_f terms. Proposition 6.20 proves the correctness of λ-splitting.

Proposition 6.12 *Let M be λ_{fc} term that represents a De Bruijn term, and let N be its $E_c \cup F_c$ normal form. Then, the following diagram commutes.*

$$\begin{array}{ccccc} M & & \xrightarrow{\beta_{fc}} & & M' \\ \Big\downarrow{\scriptstyle E_c,F_c} & & & & \Big\downarrow{\scriptstyle E_c,F_c} \\ N & \xrightarrow[\beta_{fc}]{} & N'' & \xrightarrow[E_c,F_c]{} & N' \end{array}$$

The proof of the proposition proceeds along the similar lines as the proof of Theorem 5.17 for λ_f calculus. First, we give a series of lemmas, which are λ_{fc} versions of lemmas used in proving Theorem 5.17.

Lemma 6.13 *Let M be a λ_{fc}-term. Then, the following diagram commutes provided the $E_c \cup F_c$ redex doesn't interfere with the β_{fc} redex and is not inside the abstraction part of the β_f redex.*

$$\begin{array}{ccc} M & \xrightarrow{\beta_f} & M' \\ \Big\downarrow{\scriptstyle E_c,F_c} & & \Big\downarrow{\scriptstyle E_c,F_c} \\ N & \xrightarrow[\beta_f]{} & N' \end{array}$$

Proof. The proof is by case analysis on the relative positions of the two redexes and is similar to the proof of Lemma 5.19. □

The next two lemmas show some properties of the substitution operation.

Lemma 6.14 *If* $f \underset{F}{\twoheadrightarrow} f'$, *then* $Subst[M, N, n, f] = Subst[M, N, n, f']$.

Proof. The proof is by induction on the structure of M and is similar to the proof of Lemma 5.22. The new cases are when M is $cond(f, N, \Box)$ or $cond(f, \Box, N)$. In these cases, the result follows easily from the induction hypothesis and the definition of the substitution operation. □

Lemma 6.15 *Let M and N be λ_{fc} terms. Then,*

$$Subst[M, N, 0, Id] \quad = \quad Subst[M, N, 0, P(f)]$$

for any $f \in \mathbf{F}_c$.

Proof. By induction on the structure of M, we show that

$$Subst[M, N, n, g \circ P^n(Id)] \quad = \quad Subst[M, N, n, g \circ P^n(P(f))]$$

for any f and g. Then, to prove the result, we take $g = Id$, $n = 0$ and use Lemma 6.14. The rest of the proof is similar to the proof of Lemma 5.21. The new cases are when M is a conditional. In these cases, the result follows easily from the induction hypothesis and the definition of the substitution operation. □

The next lemma shows that the substitution operation commutes with $E_c \cup F_c$ reductions. It will be used in the case when $E_c \cup F_c$ redexes are inside the abstraction part of the β_{fc} redex.

Lemma 6.16 *Let M be a λ_{fc}-term. Then,*

$$M \underset{E_c, F_c}{\longrightarrow} M' \quad \Rightarrow \quad M[N] \underset{E_c, F_c}{\longrightarrow} M'[N]$$

Proof. The proof is similar to the proof of Lemma 5.22. By induction on the structure of M, we show that

$$M \underset{E,F}{\longrightarrow} M' \quad \Rightarrow \quad Subst[M, N, n, f] \underset{E,F}{\longrightarrow} Subst[M', N, n, f]$$

1. $M \equiv i$. As in Lemma 5.22.

2. $M \equiv R\,S$. The new cases are that M is an instance of the (Ap-cL) or (Ap-cR) rule. We show the result only for the (Ap-cL) rule. Then, $M \equiv cond(g, R', \Box)\; S$, and

$$
\begin{aligned}
LHS &=_{def} cond(f, Subst[R', N, n, f], \Box)\, Subst[S, N, n, f] \\
&\underset{Ap-cL}{\longrightarrow} cond(f, Subst[R', N, n, f]\, Subst[S, N, n, f], \Box) \\
&=_{def} RHS.
\end{aligned}
$$

3. $M \equiv \lambda.R$. As in Lemma 5.22.

4. $M \equiv \varepsilon(R, g)$. The new cases are that M is an instance of the (ε-cL) or (ε-cR) rule. We show the result only for the (ε-cL) rule. Then, $M \equiv \varepsilon(cond(h, R', \Box), g)$, and

$$
\begin{aligned}
LHS &=_{def} \varepsilon(cond(h, Subst[R', N, n, g \circ f], \Box), g) \\
&\underset{\varepsilon-cL}{\longrightarrow} cond(h \circ g, \varepsilon(Subst[R', N, n, g \circ f], g), \Box) \\
&=_{def} RHS.
\end{aligned}
$$

5. $M \equiv cond(g, R, \Box)$ or $M \equiv cond(g, \Box, R)$. The only interesting cases are when M is an instance of a rule in (C-*) family. We show the result only for the (C-L) rule. Then, $M \equiv cond(L, R, \Box)$, and

$$
\begin{aligned}
LHS &=_{def} cond(L, Subst[R, N, n, f], \Box) \\
&\underset{C-L}{\longrightarrow} Subst[R, N, n, f] \\
&=_{def} RHS.
\end{aligned}
$$

This completes the proof of the lemma. □

The next lemma holds only for λ_{fc} terms that represent De Bruijn terms.

Lemma 6.17 *Let M be a λ_{fc} term that represents a De Bruijn term and is in $E_c \cup F_c$ normal form. Then,*

$$\varepsilon(M[N], \langle f, g \circ h\rangle) =_{E,F} \varepsilon(M[\varepsilon(N, g)], \langle f, h\rangle)$$

Proof. Since M represents a De Bruijn term and is in $E_c \cup F_c$ normal from, it is also a λ_f term. With this observation, the proof is identical to the proof of Proposition 5.23. □

Now, we show that Proposition 6.12 holds even if the redexes interfere.

Lemma 6.18 *Let $M \equiv \varepsilon((\lambda.R)\, S, f)$ be a λ_{fc} term that represents a De Bruijn term. Then, the following diagram commutes:*

$$
\begin{array}{ccccc}
M \equiv \varepsilon(\underline{(\lambda.R)\,S}, f) & & \xrightarrow{\quad\beta_{fc}\quad} & & M' \\
\Big\downarrow\!\!\downarrow {\scriptstyle e\text{-}ap,e\text{-}\lambda} & & & & \vdots \\
(\lambda.\varepsilon(R, P(f)))\,\varepsilon(S, f) & & & & \vdots\, {\scriptstyle E_c,F_c} \\
\Big\downarrow\!\!\downarrow {\scriptstyle E_c,F_c} & & & & \downarrow\!\!\downarrow \\
N \equiv (\lambda.R')\,\varepsilon(S, f) & \dashrightarrow_{\beta_{fc}} & N'' & \dashrightarrow\!\!\!\!\rightarrow_{E_c,F_c} & N'
\end{array}
$$

where R' is the $E \cup F$ normal form of $\varepsilon(R, P(f))$.

Proof. Similar to the proof of Lemma 5.24 using Propositions 6.6 in place of Proposition 5.12 and Lemmas 6.15, 6.16 and 6.17 in place of Lemmas 5.21, 5.22 and 5.23, respectively. □

Proof of Proposition 6.12: Similar to the proof of Theorem 5.17 using Propositions 6.6 in place of Proposition 5.12 and Lemmas 6.13, 6.16 and 6.13 in place of Lemmas 5.19, 5.22 and 5.19, respectively. □

Proposition 6.19 *Let $M \equiv C[(\lambda.R)\,S]$ be a λ_{fc} term that represents a De Bruijn term and is in $E_c \cup F_c$ normal form. Thus, M is also a λ_f term that represents a De Bruijn term and is in in $E \cup F$ normal form. Then, the following diagram commutes.*

$$
\begin{array}{ccc}
M & \xrightarrow{\quad\beta_{fc}\quad} & M' \\
\Big\downarrow\!\!\downarrow {\scriptstyle \beta_f} & & \vdots\, {\scriptstyle E_c,F_c} \\
N & \xrightarrow[\quad E,F\quad]{} & N'
\end{array}
$$

where β_f and $\beta_f c$ reductions are of displayed redex in M.

Proof. We have

$$
\begin{array}{rcll}
M' & \equiv & C[\varepsilon(R[\varepsilon(S, Env)], \langle Id, Empty\rangle)] & \\
 & =_{E_c,F_c} & C[\varepsilon(R[\varepsilon(\varepsilon(S, Env), Empty)], \langle Id, Id\rangle)], & \text{by Lemmas 6.14 and 6.17} \\
 & =_{E_c,F_c} & C[\varepsilon(R[\varepsilon(S, Id)]\langle Id, Id\rangle)] &
\end{array}
$$

Now, we claim that $\varepsilon(S, Id) \underset{E_c,F_c}{\twoheadrightarrow} S$. To prove the claim, we show by induction on the structure of S that $\varepsilon(S, P^m(Id)) \underset{E_c,F_c}{\twoheadrightarrow} S$. Basically, the result follows from the fact that $k! \circ P^m(Id) \twoheadrightarrow k!$, for any k (see Proposition 5.5). Then,

$$M' \quad =_{E_c,F_c} \quad N, \text{ by the definition of the } (\beta_f) \text{ rule.}$$

Thus, there exists a N' such that $M \underset{E_c,F_c}{\twoheadrightarrow} N' \underset{E_c,F_c}{\twoheadleftarrow} N$. But, N is a λ_f term. Thus, an $E_c \cup F_c$ reduction beginning with N is actually a $E \cup F$ reduction. The result follows. □

Proposition 6.20 *Let M be a λ_{fc} term that represents a De Bruijn term. Suppose $M \longrightarrow M'$ by an application of either the (λ-spL) or the (λ-spR) rule. Then, both M and M' have the same $E_c \cup F_c$ normal form.*

Proof. We give the proof for the (λ-spL) rule; the proof for the other rule is similar. Let $M \equiv C[\lambda.R]$ where the displayed λ-abstraction is the one to which the rule is applied. Consider the $E_c \cup F_c$ reduction of M to normal form. The reduction can be decomposed as follows:

$$M \equiv C[\lambda.R] \overset{in\ C}{\twoheadrightarrow} C'[\lambda.R] \equiv T \twoheadrightarrow N$$

Suppose $T \underset{\lambda-spL}{\twoheadrightarrow} T'$. We claim that $M' \underset{E_c,F_c}{\twoheadrightarrow} T'$. The proof of the claim is similar to the proof of Lemma 6.13.

Now, we distinguish two cases. Either $C'[]$ contains an ε term that encloses the hole, or it doesn't. We consider the first case, which is more general. Then,

$$\begin{aligned} T' &\equiv C''[\varepsilon(\lambda.R, f)] \\ &\twoheadrightarrow C''[\lambda.\varepsilon(R, P(f))] \\ &\twoheadrightarrow C''[\lambda.R'] \equiv N \end{aligned}$$

where R' is the $E_c \cup F_c$ normal form of $\varepsilon(R, P(f))$. Now, Proposition 6.6 implies that R' is a λ_{fc} term that represents a De Bruijn term.

Now, consider T'. We will write S for $cond(Id, \varepsilon(i, Pop))$,

$$\begin{aligned} T' &\equiv C''[\varepsilon(\lambda.\varepsilon(R[S], \langle Fst, L \circ Snd\rangle), f)] \\ &=_{E,F} C''[\lambda.\varepsilon(R[S], P(f) \circ \langle Fst, L \circ Snd\rangle)], \\ &\qquad \text{by the } (\varepsilon\text{-}\lambda) \text{ rule and the definition of } P, \\ &=_{E,F} C''[\lambda.\varepsilon((\varepsilon(R, P(f))[S], \langle Fst, L \circ Snd\rangle))], \\ &\qquad \text{by the definition of } Subst \text{ and Lemma 5.21,} \\ &=_{E,F} C''[\lambda.\varepsilon(R'[S], \langle Fst, L \circ Snd\rangle)], \\ &\qquad \text{by Lemma 5.22,} \\ &=_{E,F} C''[\lambda.\varepsilon(R'[\varepsilon(S, L)], \langle Fst, Snd\rangle)], \\ &\qquad \text{by Lemma 5.23,} \\ &=_{E,F} C''[\lambda.\varepsilon(R'[\varepsilon(i, Id))], \langle Fst, Snd\rangle)] \end{aligned}$$

Now, we claim that $\varepsilon(S, \langle Fst, Snd\rangle) \underset{E_c,F_c}{\twoheadrightarrow} S$. The proof is similar to that for $\varepsilon(S, Id)$; see Proposition 6.19. Also, R' represents a De Bruijn term. Thus, each occurrence of i is in the form $\varepsilon(i, f)$. Then, it is easy to show that $R'[\varepsilon(i, Id)] \underset{E_c,F_c}{\twoheadrightarrow} R'$.

Thus, $T' =_{E_c,F_c} N$. The result follows by noticing that N is in $E_c \cup F_c$ normal form; so, T' reduces to N. □Now, we can give the proof of Theorem 6.11.

Proof of Theorem 6.11:

Part 1. For $E_c \cup F_c$ rules, both claims follow from their confluence. For (λ-spL) and (λ-spR), both claims follow from Proposition 6.20.

Part 2. Both claims follow from Proposition 6.12, Proposition 6.19, and the correspondence between λ_f calculus and De Bruijn notation, *i.e.*, Theorem 5.25. □

6.3 Term approximations of term graphs

In this section, we formulate the notion of terms (*i.e.*, finite trees) obtained by unraveling a term graph and replacing some subtrees by □ nodes. The notion will be used to relate λ_{fc} terms and λ_{fc} graphs.

Graphs in this section mean term graphs over a signature Σ, and terms mean terms over Σ augmented with a function symbol □ of arity 0. We will consider terms as trees and use the same notation as for graphs.

The following is the main definition.

Definition 6.21 Let $G = (N, lab, succ, r)$ be a graph, and let $T = (N_T, lab_T, succ_T, r_T)$ be a tree. T *approximates* G iff there exists a function $\mu : N_T \rightarrow N$ such that

$$\mu(r_T) = r$$

and for any node $n \in N_T$ that is not labelled by □

$$\begin{aligned} lab(\mu(n)) &= lab_T(n) \\ succ(\mu(n)) &= \mu(succ_T(n)) \quad \text{where } \mu(n_1, \ldots, n_k) = (\mu(n_1), \ldots, \mu(n_k)). \end{aligned}$$

That is, μ maps root to root and preserves labels, successors and their order for all nodes in the tree except for the ones labelled by □. The function μ will be called an *approximating morphism* from T to G.

Notice that an approximating morphism acts like a homomorphism from an open term graph to a term graph (hence the name); nodes labelled with □ act like open nodes.

If a term approximates a graph, then the conditions to be satisfied by an approximating morphism imply that it is unique. Putting it another way, there is only one way to unravel a graph and place □ nodes to get a given term.

Proposition 6.22 *If a term approximates a graph, then there is an unique approximating morphism from the term to the graph.*

Proof. By contradiction. Suppose μ and μ' are two distinct approximating morphisms from the term to the graph. Then, there must be a node, say n, in the tree such that $\mu(n) \neq \mu'(n)$. Node n cannot be the root of the tree, since both μ and μ' map the root of the tree to the root of the graph. Now, we can choose n so that μ and μ' agree on all other nodes on the path from the root of the tree to n. Let m be the node immediately before n on the path. Now, $\mu(m) = \mu'(m)$ and both μ and μ' preserve successors and their order. Thus, $\mu(n) = \mu'(n)$. Contradiction. □

Remark 6.23 Let T be an approximant of G, and suppose μ is the approximating morphism from T to G. Then, Definition 6.21 implies the following.

1. If $(n_1, i_1, \ldots, n_k)$ is a rooted path of T, then there is a corresponding rooted path in G given by $(\mu(n_1), i_1, \ldots, \mu(n_k))$ Moreover, labels of the corresponding nodes on the two paths are identical with the usual exception for □ node.

2. If $(n_1, i_1, \ldots, n_k)$ is a rooted path of G, then one of the following holds.

   - There is a rooted path $(n'_1, i_1, \ldots, n'_k)$ in T such that $\mu(n'_j) = n_j$, for $1 \leq j \leq k$ and labels of the corresponding nodes on the two paths are identical.

   - There is a rooted path $(n'_1, i_1, \ldots, n'_l)$ in T with $l \leq k$ such that $\mu(n'_j) = n_j$ for $1 \leq j \leq l$, and labels of the corresponding nodes on the two paths are identical except that the label of n'_l is □.

The following proposition shows that the approximation relation carries over to subterms.

Proposition 6.24 *Let G be a λ_{fc} graph, and let T be an approximant of G with μ being the approximating morphism. For any node n of T, $T|n$ (the subtree rooted at n) approximates $G|\mu(n)$.*

Proof. Let μ' be μ restricted to the nodes of $T|n$. We show that μ' is an approximating morphism from $T|n$ to $G|\mu(n)$.

The main part is to show that the range of μ' is a subset of the node set of $G|\mu(n)$. Let n' be a node in $T|n$. Then, the tree T contains a rooted path $r_T, \ldots, n, \ldots, n'$. By Remark 6.23, there is a rooted path $\mu(r_T), \ldots, \mu(n), \ldots \mu(n')$ in G. Thus, there is a path from $\mu(n)$ to $\mu(n')$. Since $\mu(n') = \mu'(n')$, there is a path from the root of $G|\mu(n)$ to $\mu'(n')$. Hence $\mu'(n')$ is a node of the subgraph $G|\mu(n)$.

It is easy to check that μ' satisfies the remaining conditions required of an approximating morphism as it is the restriction of μ to nodes of $T|n$. □

The following proposition relates trees obtained by unraveling (acyclic) graphs and a special type of approximants of graphs. See [7] for a definition of the tree obtained by unraveling a graph.

Proposition 6.25 *The tree obtained by unraveling a graph approximates the graph.*

As a partial converse, if a tree approximates a graph and contains no □ nodes, then the tree is identical to the tree obtained by unraveling the graph.

Proof. There exists a rooted homomorphism (*i.e*, a homomorphism that maps root to root) from the tree obtained by unraveling the graph to the graph (see [7]). Now, a rooted homomorphism from a tree to a graph is also an approximating morphism from the tree to the graph. The result follows.

To prove the converse, suppose μ is the approximating morphism from the tree to the graph. Now, the tree contains no □ nodes. (So μ is also a rooted homomorphism from the tree to the graph.) Then, Remark 6.23 implies that $(n_1, i_1, \ldots, n_k)$ is a path in the tree *iff* $(\mu(n_1), i_1, \ldots, \mu(n_k))$ is path in the graph. Furthermore, the labels of corresponding nodes on the two paths are identical. This implies that the tree is identical to the tree obtained by unraveling the graph. □

Corollary 6.26 *If G has an approximant that contains no □ nodes, then G is acyclic.*

The next proposition will be used to relate reduction of graphs to reduction of terms. It shows how the redirection of pointers changes approximations. First, a notation.

Notation: Let $T, T_1, \ldots T_k$ be trees. Let $n_1, \ldots, n_k$ be nodes of T such that subtrees rooted at these nodes are pairwise disjoint. Then, $T[n_1 := T_1, \ldots, n_k := T_k]$ denotes the tree that is obtained by replacing n_i by T_i for $1 \leq i \leq k$.

Proposition 6.27 *Suppose the following:*

1. *G is a term graph, and n and n' are two distinct nodes of G;*
2. *$G' = G[n := n']$;*
3. *m is a node of G, not necessarily distinct from n or n';*
4. *T approximates the subgraph $G|m$ with μ being the approximating morphism;*
5. *$\mu^{-1}(n)$ is not empty, and $n_1, \ldots, n_k$ are all those nodes of $\mu^{-1}(n)$ for which the path from the root of T to the node doesn't include a node in $\mu^{-1}(n)$;*
6. *$T_1, \ldots T_k$ are trees, each of which approximates $G'|n'$.*

Then, $T[n_1 := T_1, \ldots, n_k := T_k]$ approximates $G'|\phi(m)$ where

$$\phi(m) = \begin{cases} n', & \text{if } m = n \\ m, & \text{otherwise} \end{cases}$$

If $\mu^{-1}(n)$is empty, then we simply have that T approximates $G'|\phi(m) = G'|m$.

Proof. Let $T' \equiv T[n_1 := T_1, \ldots, n_k := T_k]$. First, we carry out the proof assuming that $m = r_G$ where r_G is the root of G. There are two cases.

First, suppose $r_G = n$. Then, the root of T is one of $n_1, \ldots, n_k$, and hence, T' is simply one of $T_1, \ldots, T_k$. The result follows from the assumption that each of $T_1, \ldots, T_k$ approximates $G'|n'$.

So suppose $r_G \neq n$. Suppose μ is the approximating morphism from T to G, and suppose, for $1 \leq i \leq k$, μ_i is the approximating morphism from T_i to $G'|n'$. We construct an approximating morphism from T' to G'. Let

$$N_- = (N_T - N_{T|n_1} - \cdots - N_{T|N_k}).$$

Then, the node set of T' is given by

$$N_T' = N_- \cup N_{T_1} \cup \cdots \cup N_{T_k}$$

Let $\mu' : N_{T'} \to N_{G'}$ be the function defined as follows:

$$\mu'(p) = \begin{cases} \mu(p), & \text{if } p \in N_- \\ \mu_i(p), & \text{if } p \in N_{T_i}, \text{ for } 1 \leq i \leq k. \end{cases}$$

We show that μ' is an approximating morphism from T' to G'. The root of T' is the root of T, and the root of G' is the root of G. Furthermore, the root of T' is in the set N_- on which μ and μ' agree. Then, μ maps the root of T to the root of G implies that μ' maps the root of T' to the root of G'. μ' preserves labels follows simply from its definition and the fact that $\mu, \mu_1, \ldots, \mu_k$ preserve labels. Finally, we have to show that μ' preserves successors and their order. First consider a node of T' that it is a node of T_i. Successors of such a node are also nodes of T_i. Since μ' and μ_i behave identically on nodes of T_i and since μ_i preserves successors and their order, so does μ'. Now, consider a node p of T' that is in the set N_-. The jth successor of p is either in the same set, in which case the result follows from the fact that μ and μ' agree on this set. Or, the successor is the root of one of $T_1, \ldots, T_k$. Then, in T, the jth successor of p must be one of $n_1, \ldots, n_k$. Thus, the jth successor of $\mu(p)$ in G must be n, and hence the jth successor of $\mu(p)$ in G' is n'. Since, μ' maps the roots of $T_1, \ldots, T_k$ to n', the result follows.

Thus T' approximates G'. By Proposition 6.24, T' also approximates $G'|r_{G'}$. Since $r_{G'} = r_G = \phi r_G$, the result follows.

Now, we consider the case when m is not the root of the graph. We proceed as above but starting with a graph that is identical to G except that the root of the graph is m.

If $\mu^{-1}(n)$ is empty, then the result that T approximates $G'|m$ follows simply from the above proof. □

6.4 Proof of Correctness

In this section, we give proof of the correctness of the interpreter using the approach described in the introduction.

The motivation for the notion of term approximations to graphs discussed in the last section was to formalize the intuitive idea of λ_{fc} terms obtained by unraveling λ_{fc} graphs and replacing one side of each conditional node by □. There is, however, a minor problem in relating the notion of λ_{fc} graphs to λ_{fc} terms using the notion of approximations. In λ_{fc} graphs, the function symbol λ is used with arity 2 because of explicit binding pointer between λ nodes and i nodes. On the other hand, the function symbol λ in λ_{fc} terms has arity 1. To relate λ_{fc} terms and graphs, we essentially ignore binding pointers in graphs and consider λ to be of arity 1. The effect of binding pointers is captured separately. Note that λ_{fc} terms are a particular type of approximants of λ_{fc} graphs. In this section, we will consider only those approximants of λ_{fc}

graphs that are λ_{fc} terms, and thus, we will usually omit the qualification that approximants are λ_{fc} terms.

The notion of approximations will also be used to relate functions on control environments and their graph representation. Notice that functions on control environments don't contain $\Box$ symbols. Thus, a function graph has exactly one approximant, which is the term obtained by unraveling the graph (see Proposition 6.25). Again note that we are considering only a particular type of approximants of function graphs.

The following proposition shows the basic relationship between different approximants of a λ_{fc} graphs.

Proposition 6.28 *Suppose T and T' are different λ_{fc} terms, both of which approximate a λ_{fc} graph. Then, they must be of the form*

$$T \equiv C[S_1, S_2, \ldots, S_k] \text{and} \;\; T' \equiv C[S'_1, S'_2, \ldots, S'_k]$$

where C is a context with k holes and, for $1 \leq j \leq k$, S_j and S'_j are related in the following way:

$$S_j \equiv cond(f, R, \Box), \quad S'_j \equiv cond(f, \Box, R')$$

or the same but with the position of $\Box$ interchanged.

Proof. By induction on the structure of T (or T').

$T \equiv i$. The root of the graph must be labelled with i. Thus, the graph has only one approximant, and the result is vacuously true.

$T \equiv T_1\, T_2$. The root of the graph must be labelled with ap. Since T' is an approximant of the graph, it must be of the form $T'_1\, T'_2$. Now, $T \not\equiv T'$ implies one of the following: $T_1 \not\equiv T'_1$, $T_2 \not\equiv T'_2$, or both. Assume the last case as the other two are special cases of the last. Now, by Proposition 6.24, both T_1 and T'_1 are approximants of the first successor of the root. So by induction hypothesis T_1 and T'_1 must be of the form

$$T_1 \equiv C_1[U_1, U_2, \ldots, U_{k_1}] \text{and} \;\; T'_1 \equiv C_1[U'_1, U'_2, \ldots, U'_{k_1}]$$

where C_1 is the common context with k_1 holes and terms filling the holes satisfy the required relation. By a similar argument, T_2 and T'_2 must be of the form.

$$T_2 \equiv C_2[V_1, V_2, \ldots, V_{k_2}] \text{and} \;\; T'_2 \equiv C_2[V'_1, V'_2, \ldots, V'_{k_2}]$$

where C_2 is a context with k_2 holes and terms filling the holes satisfy the required relation. To write T and T' in the required form, we let $C \equiv C_1[\ ,\ldots,\ ]\,C_2[\ ,\ldots,\ ]$ with $k = k_1 + k_2$ holes. Then,

$$T \equiv C[U_1, \ldots, U_{k_1}, V_1, \ldots V_{k_2}] \text{and}$$
$$T' \equiv C[U'_1, \ldots, U'_{k_1}, V'_1, \ldots V'_{k_2}]$$

and terms filling the holes satisfy the required relation. This proves the result.

$T \equiv \lambda.T_1$. The proof is similar to the case of application.

$T \equiv \varepsilon(T_1, f)$. Then, T' must be of the form $\varepsilon(T'_1, f')$; the argument is similar to the one used in the case of application. By Proposition 6.24, both f and f' approximate the subgraph rooted at the second successor of the root of the graph. Since f and f' contain no $\square$ nodes, Proposition 6.25 implies that $f \equiv f'$. The rest of the proof is similar to the application case.

$T \equiv cond(f, T_1, \square)$. Again, T' must be a conditional. Moreover, function parts of T and T' must be identical (see the εcase). Thus, either $T' \equiv cond(f, T'_1, \square)$ or $T' \equiv cond(f, \square, T'_1)$. In the first case, the rest of the proof is similar to that for application. In the second case, the result follows by taking the common context to be a hole.

$T \equiv cond(f, \square, T_1)$. Similar to the above case. $\square$

Now intuitively, if a λ_{fc} graph can be translated to a De Bruijn term (see Chapter 4 for the procedure), then the set of its approximants contains a λ_{fc} term that represents a De Bruijn term. The following proposition shows that this term is unique. Basically, either there is no way to place $\square$ nodes in the tree obtained by unraveling a λ_{fc} graph so that the resulting tree is finite and $\square$ nodes are always on the useless sides of conditionals, or there is exactly one way to do so.

Proposition 6.29 *For any λ_{fc} graph, the set of its approximants contains at most one λ_{fc} term that represents a De Bruijn term.*

Proof. By contradiction. Suppose T and T' are different approximants of a λ_{fc} graph, and suppose both represent De Bruijn terms. By Proposition 6.28, T and T' are of the form

$$T \equiv C[S_1, S_2, \ldots, S_k] \text{and} \quad T' \equiv C[S'_1, S'_2, \ldots, S'_k]$$

where C is a context with k holes and, for $1 \leq j \leq k$, S_j and S'_j are conditionals with the same function part but with $\square$ on different sides. Then, an easy extension of Proposition 6.8 implies that both T and T' cannot represent De Bruijn terms. Contradiction. $\square$

Now, we define λ_{fc} graphs that we consider valid representation of De Bruijn terms. The interpreter operates only on these graphs.

Definition 6.30 A λ_{fc} graph represents a De Bruijn term iff the following holds.

1. The set of its approximants contains a λ_{fc} term that represents a De Bruijn term.

2. Suppose T is the λ_{fc} term that represent a De Bruijn term and approximates the graph with μ being the approximating morphism. Let n be a λ node in G, and let n' be an i node in G. Let

$$N_\lambda = \{m \mid m \in \mu^{-1}(\nu(n)) \text{ and } lab(m) \neq \Box\}$$
$$N_i = \{m \mid m \in \mu^{-1}(\nu(n')) \text{ and } lab(m) \neq \Box\}.$$

Then, node n has a binding pointer to node n' iff N_i contains all and only those occurrences of i that are bound by λ-abstractions in N_λ.

The second condition in the definition ensures that the use of binding pointers in λ_{fc} graphs that represent De Bruijn terms is consistent with the notion of bound occurrences of i in λ_{fc} calculus (or the corresponding notion in De Bruijn notation). Furthermore, there is no unintentional sharing of i's, *e.g.*, an i node in a λ_{fc} graph that represents a De Bruijn term doesn't represent both a free occurrence of i in T and a bound occurrence of i in T.

Definition 6.31 Let G be a λ_{fc} graph that represents a De Bruijn term. Then,

1. $LFC(G)$ denotes the unique λ_{fc} term that represents a De Bruijn term and matches G.

2. $Tr(G)$ denotes the De Bruijn expression obtained as follows. Translate $LFC(G)$ to a λ_f term and then translate the λ_f term to a De Bruijn term.

6.4.1 Correspondence between the graph reduction system underlying the interpreter and λ_{fc} calculus

In this section, we show the precise correspondence between reduction of λ_{fc} graphs and reduction of λ_{fc} terms. The main theorem is the following.

Theorem 6.32 *Let G be a λ_{fc} graph that represents a De Bruijn term, and let $G \xrightarrow[I]{} G'$. Then,*

1. G' is a λ_{fc} graph that represents a De Bruijn term.

2. (a) *If* $G \underset{\beta_{fc}}{\longrightarrow} G'$, *then* $LFC(G) \underset{\beta_{fc}}{\twoheadrightarrow} LFC(G')$.

   (b) *If* $G \underset{\neg\beta_{fc}}{\longrightarrow} G'$, *then* $LFC(G) \underset{\neg\beta_{fc}}{\twoheadrightarrow} LFC(G')$.

   Moreover, in both cases the reduction on λ_{fc} terms is an innermost reduction of a set of redexes in $LFC(G)$.

The proof of the theorem will be given after a series of lemmas. First, we show the correspondence between reduction of functions on control environments and reduction of F_c graphs. Intuitively, the correspondence between the two is a simple consequence of the following facts. First, F_c graphs are acyclic. Second, functions on control environments don't contain □ symbol; so if a function on control environments approximates a F_c graph, it is the tree obtained by unraveling the graph. Third, the graph reduction rules (the set F_g) are derived from the term rules (the set F_c) by using the standard technique for converting left-linear term rule to graph rules[2].

Lemma 6.33 *Let G be a F_c graph, and suppose $G \underset{F_g}{\longrightarrow} G'$. Let T be a term in $\mathbf{F}_c$ that approximates G. Then,*

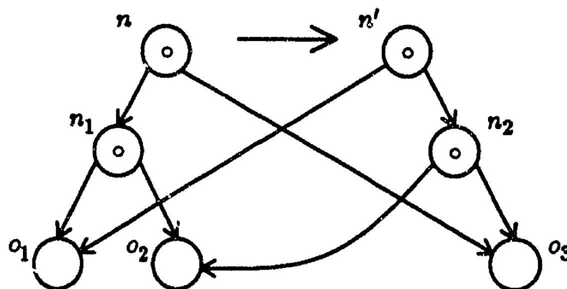
1. *All subtrees of T whose roots map onto the root of the redex in G are F_c redexes. Furthermore, these redexes are pairwise disjoint.*

2. *Suppose $T \underset{F_c}{\twoheadrightarrow} T'$ by a complete (innermost) reduction relative to the redexes identified in (1). Then, T' approximates G'.*

Proof. By case analysis on the rules in F_g. Note the all the rules in F_g are of the form (H, n, n') where H is an open F_c graph and n, n' are nodes of H. We will prove the lemma for the (Ass) rule. The proof for other rules is similar and is omitted.

The graph rule is shown below; we have given names to all nodes in order to refer to them easily.

(Ass):



[2]The lemma holds in general under these conditions

Let ν be the occurrence of this rule that is reduced in the reduction $G \longrightarrow G'$. Recall that ν is a homomorphism from $H|n$ to G and preserves labels, successors and their order except at empty nodes. Let μ be the approximating morphism from T to G. Let $N = \mu^{-1}(\nu(n))$; N is the set of all nodes of T that map onto n. Note that N cannot be empty.

Part 1. First, we show that the subtree rooted at any node in N is an instance of the left hand side of the term version of the (Ass) rule given below

(Ass): $\quad (f \circ g) \circ h \longrightarrow f \circ (g \circ h)$

Let $m \in N$, and let $R = T|m$. Now, ν preserves labels, successors and their order for all nodes of $H|n$ except empty nodes, and μ preserves labels, successors and their order for all nodes of T. Note that T contains no $\Box$ nodes. So

$$lab_H(n) = lab_G(\nu(n)) = lab_T(m) = \circ .$$

In T, let m' be the first successor of m. Then,

$$\nu(n_1) = succ_G(\nu(n))_1 = \mu(m').$$

So again,

$$lab_H(n_1) = lab_G(\nu(n_1)) = lab_T(m') = \circ .$$

Thus, R is of the form $(f \circ g) \circ h$ for some functions on control environments f, g and h. Therefore, R is an instance of the left hand side of the term rule.

To prove that redexes rooted at nodes in N are pairwise disjoint, let m_1 and m_2 be nodes in the set N. Now, assume that the redex rooted at m_1 includes the redex rooted at m_2. Then, there is a path $m_1, \ldots, m_2$ in T, and hence, there is a path $\mu(m_1), \ldots, \mu(m_2)$ in G (see Remark 6.23). But $\mu(m_1) = \mu(m_2)$; so G contains a cycle, contradicting the fact that G is acyclic.

Part 2. The graph G' is given by (see Section 4.3)

$$\begin{aligned} A &= G +_\nu (H, n, n') \\ B &= A[\nu(n) := n'] \\ G' &= GC(B) \end{aligned}$$

We proceed in three steps.

1. We show the following. Let T_1 be a subtree of T such that T_1 includes no node in N. Suppose μ maps the root of T_1 to the node p in G. Then, S approximates $B|p$. The proof is as follows.

   Let A_r be the subgraph of A rooted at its root. By the definition of $+_\nu$, A_r is identical to G. So T also approximates A_r with μ being the approximating morphism. Then, by Proposition 6.24, T_1 approximates $A_r|p$, which is identical to $A|p$. Since T_1 contains no node of N, μ maps no node of T_1 to the node $\nu(n)$ in A_r. Therefore, the approximating morphism from T_1 to $A|p$, which is simply restriction of μ to the nodes of T_1, maps no node of T_1 to $\nu(n)$. By applying Proposition 6.27, we get T_1 approximates $B|p$.

   Note that the node p may be part of the pattern of the redex in G, *i.e.*, the first successor of $\nu(n)$. The claim shows that the subterm represented by the subgraph rooted at p along any pointer doesn't change after redirection of pointers. (This is true even along a pointer from the node $\nu(n)$ of B.)

2. Let $m \in N$, and let $R = T|m$. Then,

$$\begin{aligned} R &\equiv (f \circ g) \circ h \\ &\xrightarrow[Ass]{} f \circ (g \circ h) \equiv R' \end{aligned}$$

   We show that R' approximates $B|n'$. Consider the term f. Preservation of successors and their order by ν (on non-empty nodes) and by μ implies that μ maps the root of f to the node $\nu(o_1)$ of G. Also, f contains no node in N. By (a) above, f approximates $B|\nu(o_1)$. Similarly, g and h approximate $B|\nu(o_2)$ and $B|\nu(o_3)$, respectively. Also note that, in B, $\nu(o_1)$ is the first successor of n', $\nu(o_2)$ is the first successor of n_2, and $\nu(o_3)$ is the second successor of n_2 . Now, we define $\mu' : N'_R \to N_{B|n'}$ in the following way. It maps the root of R' to n' and the second successor of the root of R' to the n'_1. For all nodes in f, it simply mimics the approximating morphism from f to $B|\nu(o_1)$. Similarly, for nodes in g and h. Then, it is easy to verify that μ' is an approximating morphism from R' to $B|n'$.

3. To complete the proof, let $N = \{m_1, \ldots m_k\}$. For $1 \leq j \leq k$, let R'_j be the reduct of the redex rooted at m_j. Then, $T' = T[m_1 := R'_1, \ldots m_k = R'_k]$. As argued in (a), T approximates $A|r_A$. As shown in part (b), each of $R'_1, \ldots, R'_k$ approximates $B|n'$. Using Proposition 6.27, we conclude that T' approximates $B|\phi(r_A)$. By the definition of ϕ and

the definition of the operation of redirecting pointers, $\phi(r_A) = r_B$. So T' approximates $B|r_B = GC(B) = G'$.

This completes the proof. □

The definition of λ_{fc} graphs, see Definition 4.21, puts certain structural constraints on graphs. We verify that these constraints hold after reduction. Recall that the set of graph rules used by the interpreter is named I.

Lemma 6.34 *Let G be a λ_{fc} graph. If $G \underset{I}{\longrightarrow} G'$, then G' is a λ_{fc} graph.*

Proof. We have to check that G' satisfies the conditions of Definition 4.21.

First, we show that function parts of ε and *cond* nodes are acyclic graphs. The only rules that can change the structure of function parts are F_g rules, and the result follows from Lemma 6.33.

Second, we show that an i has a binding pointer from at most one λ node. The only way an i node can have binding pointers from two λ nodes is that the reduction of G to G' copies a λ node but not the i node bound by the λ node. There are only three rules that may copy a λ node: (ε-λ), (β_{fc}), and (λ-sp). The first two rules require that the λ node that is part of the pattern of the redex has exactly one pointer. Thus, these rules don't copy a λ node. The last rule explicitly ensures that if a λ node is copied, then i node bound by the λ node is also copied.

The rest of the conditions are easy to verify by simple inspection of the rules. □

The next four lemmas show that if G reduces to G', then $LFC(G)$ reduces to a λ_{fc} term that approximates G'.

Lemma 6.35 *Let G be a λ_{fc} graph that represents a De Bruijn term, and let $G \underset{F_g}{\longrightarrow} G'$. Then, $LFC(G) \underset{E_c}{\twoheadrightarrow} T'$ by an innermost complete reduction relative to a set of the redexes in $LFC(G)$, and T' approximates G'.*

Proof. Basically, follows from Lemma 6.33.

Lemma 6.36 *Let G be a λ_{fc} graph that represents a De Bruijn term, and let $G \underset{E_g}{\longrightarrow} G'$. Then,*

1. *All subtrees of $LFC(G)$ whose roots map onto the root of the redex in G and are not labelled with □ are E_c redexes. Furthermore, these redexes don't interfere with each other.*

2. Suppose $LFC(G) \xrightarrow[E_c]{} T'$ by an innermost complete reduction relative to the redexes identified in (2). Then, T' approximates G'.

Proof. By case analysis on the rules in E_g. Note the all the rules in E_g are of the same form as the rules in F_g, *i.e.*, they are of the form (H, n, n') where H is an open λ_{fc} graph and n, n' are nodes of H. Thus, the proof is along the lines of the proof of Lemma 6.33 with a few changes made to handle $\Box$ nodes and non-disjoint redexes.

Let ν be the occurrence of an E_g rule that is reduced in the reduction $G \longrightarrow G'$. Let μ be the approximating morphism from T to G. Let

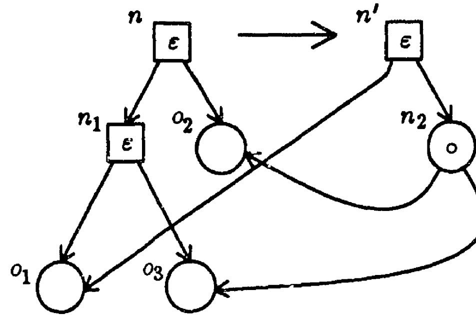
$$N = \{m \mid m \in \mu^{-1}(\nu(n)) \text{ and } lab(m) \neq \Box\}$$

Note that N can be an empty set. We will assume that N is non-empty, which is the general case.

Proof of the second part is the same for all the rules. Proof of the first part is somewhat different for different rules. We will prove both parts for the $(\varepsilon\text{-}\varepsilon)$ rule. For other rules, we will simply point out the relevant differences.

Rule $(\varepsilon\text{-}\varepsilon)$: The graph rule is shown below; we have given names to all nodes in order to refer to them easily.

$(\varepsilon\text{-}\varepsilon)$:



Part 1. First, we show that the subtree rooted at any node in N is an instance of the left hand side of the term version of $(\varepsilon\text{-}\varepsilon)$ rule given below

$$(\varepsilon\text{-}\varepsilon): \qquad \varepsilon(\varepsilon(M, f), g) \longrightarrow \varepsilon(M, f \circ g)$$

Let $m \in N$, and let $R = T|m$. Now, ν preserves labels, successors and their order for all nodes of $H|n$ except empty nodes, and μ preserves labels, successors and their order for all nodes of T except $\Box$ nodes. Proceeding as in the proof of Part (1) of Lemma 6.33, we show that R must be one of the following forms:

- $\Box$,
- $\varepsilon(\Box, f)$, or
- $\varepsilon(\varepsilon(M, f), g)$.

R cannot be of the first form as nodes in N are not labelled with $\Box$. R cannot be of the second form because a $\Box$ node cannot be a successor of a ε node; $\Box$ nodes occur only as successors of *cond* nodes. Thus, R must be of third form, which, of course, is an instance of the left hand side of the term rule.

Now, we show that redexes rooted at nodes in N don't interfere with each other. The proof is by contradiction. We will show that if two redexes rooted at nodes in N interfere, then T cannot be a λ_{fc} term. The basic idea is that if two redexes in N interfere, then the root of the redex in G contains a pointer from itself. This cycle cannot be broken by resolving conditionals, and therefore, there is no λ_{fc} term that approximates G. Let m_1 and m_2 be nodes in the set N such that the redex rooted at m_1 interferes with the redex rooted at m_2. Now, two (ε-ε) redexes interfere only if the root of one is the first successor of the other, *i.e.*, they are of the form $\varepsilon(\varepsilon(M, f), g$. So assume that m_2 is the first successor of m_1. Then, T contains a path of of the form $(m_1, i_1, m_2, \ldots, m_k)$ such that the following holds: all nodes other than m_k are labelled with ε m_k is not labelled with ε, and m_j is the first successor of m_{j-1} for $1 < j \leq k$. Now, $\mu(m_1) = \mu(m_2) = \nu(n)$. Properties of μ imply that μ maps other nodes on the path also to $\nu(n)$. Thus, label of m_k must be either ε or $\Box$. Since the label of m_k is not ε, the label must be $\Box$. But then T is not a λ_{fc} term as a $\Box$ node is the first successor of an ε node. Contradiction.

Part 2. The graph G' is given by

$$\begin{aligned} A &= G +_{\nu} (H, n, n') \\ B &= A[\nu(n) := n'] \\ G' &= GC(B). \end{aligned}$$

Now, $T \twoheadrightarrow T'$ by an innermost complete reduction relative to the redexes rooted at the nodes in N. Note that redexes reduced in the reduction are not disjoint, only non-interfering. To show that T' approximates G', we prove the following claim, which immediately implies the result.

Claim: Let T_1 be a subtree of T, and suppose μ maps the root of T_1 to the node m in G. Consider all the redexes in T_1 whose roots are in the set N, and suppose $T_1 \twoheadrightarrow T_1'$ by an

innermost complete reduction relative to these redexes. Then, T_1' approximates $B|\phi(m)$ where $\phi(m)$ is defined as follows (see Proposition 6.27):

$$\phi(m) = \begin{cases} n', & \text{if } m = \nu(n) \\ m, & \text{otherwise} \end{cases}$$

The proof is by induction on the number of nodes of T_1 that belongs to the set N.

Base Step. Suppose T_1 contains no nodes that are in the set N. Then, $T_1' \equiv T_1$ and $\phi(m) = m$. The proof that T_1 approximates $B|m$ is identical to Part 2(a) of the proof of Lemma 6.33.

Induction step. Suppose T_1 contains k nodes that are in the set N. First, suppose the root of T_1 is in N. Thus, $m = \nu(n)$ and $\phi(m) = n'$. Now,

$$T_1 \equiv \varepsilon(\varepsilon(M, f, g)).$$

Since redexes rooted at the nodes in N don't interfere, the reduction from T_1 to T_1' is of the form

$$\begin{aligned} T_1 &\equiv \varepsilon(\varepsilon(M, f, g)) \\ &\twoheadrightarrow \varepsilon(\varepsilon(M', f'), g') \\ &\longrightarrow \varepsilon(M', f' \circ g') \equiv T_1'. \end{aligned}$$

where M' is obtained from M by an innermost complete reduction relative to the redexes whose roots are in N, and f' and g' are obtained from f and g similarly. Actually, $f' = f$ and $g' = g$ as f and g don't contain any nodes that are in N. Now, T_1 approximates $G|\nu(n)$. By Proposition 6.24 and properties of μ and ν, M approximates $G|\nu(o_1)$. Also, the number of nodes in M that are in the set N is less than k. By induction hypothesis, M' approximates $B|\phi(\nu(o_1))$. By the definition of the operation of redirecting pointers, $\phi(\nu(o_1))$ is simply the first successor of the node n' in B. So M' approximates the subgraph of B that is rooted at the first successor of n'. Similarly, we show that f' and g' approximate subgraphs rooted at the first and second successor of n_2, respectively. Given this, it is easy to establish that T_1' approximates $B|n'$ (see Part 2(b) of the proof of Lemma 6.33).

Now, suppose the root of T_1 is not in the set N. Consider redexes in T_1 whose roots are in the set N and among them consider the ones that are outermost. Let $R_1, \ldots R_j$ be these redexes, and let $m_1, \ldots m_j$ be their roots. Then,

$$T_1' \equiv T_1[m_1 := R_1', \ldots, m_j := R_j']$$

and, for $1 \leq i \leq j$, $R_i \twoheadrightarrow R_i'$ by an innermost complete reduction relative to the redexes whose roots are in N. Since T_1 approximates $G|m$, it also approximates $A|m$ (see Part 2(a) of Lemma 6.33). As shown above each of $R_1, \ldots, R_j$ approximates $B|n'$. By Proposition 6.27, T_1' approximates $B|\phi(m)$.

This completes the proof of the claim.

Now, we complete the proof of Part (2). The above claim implies that T' approximates $B|\phi(r_G)$. But $\phi(r_G) = \phi(r_A) = r_B$. So T' approximates $B|r_B = GC(B) = G'$.

Rule (ε-λ): The only difference from the proof for the (ε-ε) rule is that (ε-λ) redexes never interfere with each other. Note that binding pointers play no role in the correspondence between λ_{fc} terms and λ_{fc} graphs.

The following observation will be useful in the proof of optimality. All λ nodes of T' that correspond to the λ-abstraction on the right hand side of the term rule map to the node n' in G'.

Rule (ε-ap): Again, the only difference is that (ε-ap) redexes never interfere with each other. As for the (ε-λ) rule, note that all *ap* nodes of T' that correspond to the application on the right hand side of the term rule map to the node n' in G'.

Rule (ε-c): The proof of Part (1) is as follows. Proceeding as for the (ε-ε) rule, we show that the subtree rooted at a node in N must be either

$$\varepsilon(cond(f, M, \Box), g) \quad \text{or} \quad \varepsilon(cond(f, \Box, M), g)$$

The first is an instance of the left side of the (ε-cL) rule; the second is an instance of the left hand side of the (ε-cR) rule. Note that two different term rules are used to simulate a graph rule. Now, a (ε-cL) redex never interferes with another (ε-cL) redex or a (ε-cR) redex. Similar comments apply for a (ε-cR) redex.

The proof of Part (2) is similar to that for the (ε-ε) rule.

Rule (Ap-c): Similar to the rule (ε-c). In this case, we use both (Ap-cL) and (Ap-cR) to simulate the graph rule. The following observation will be useful in the proof of optimality. For all (ε-cL) redexes in T, the *ap* nodes of T' that correspond to the application on the RHS of the rule maps to the left hand side of the conditional node n' of G'. Similarly, all *ap* nodes of T' that correspond to the RHS of the (ε-cR) rule map to the right hand side of n'.

Rules in the $C-$ family: We discuss the proof for the (C-L) rule; other rules are dealt with similarly. The proof of Part (1) is as follows. Proceeding as for the (ε-ε) rule, we show that the subtree rooted at a node in N must be either

$$(cond(L, M, \Box) \quad \text{or} \quad cond(L, \Box, M).$$

Since T is a λ_{fc} term that represents a De Bruijn term, Proposition 6.7 implies that no conditional inside T can be be of the form $cond(L, \Box, M)$. Thus, the subtree must be of the first form, which is an instance of the left hand side of (C-L) rule for λ_{fc} terms. Now, a (C-L) redex never interferes with another (C-L) redex. So Part (1) holds. The proof of Part (2) is similar to that for the (ε-ε) rule.

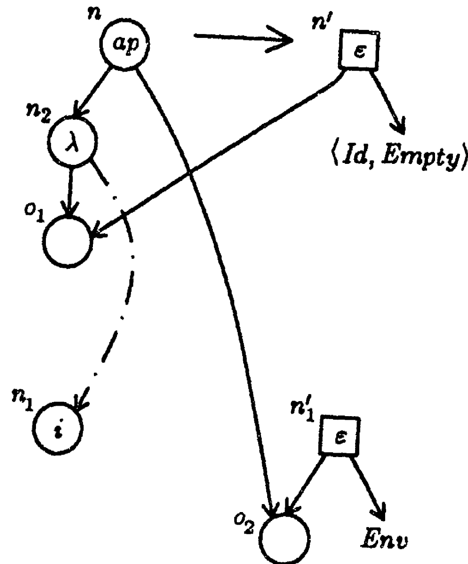
The proof of the lemma is now complete. $\Box$

Lemma 6.37 *Same as Lemma 6.36 but for the (β_{fc}) rule.*

Proof. The proof is along the lines of the proof of Lemma 6.33 but with some crucial changes made to handle redirection of pointers for two nodes (see below).

For reference, we reproduce the (β_{fc}) rule for graphs and the (β_{fc}) rule for terms. The β_{fc} rule for graphs is given below.

(β_{fc}):



No other pointers to λ-node

The rule is of the form (H, n, n', n_1, n_1') where H is the whole graph and nodes n, n', n_1, n_1' are as indicated. Note a couple of points. First, the rule is applied only if the λ node in an instance of the rule has only one pointer. Second, an application of this rule involves redirection of pointers for two nodes. The β_{fc} rule for terms is as follows.

(β_{fc}): $\quad (\lambda.M)\, A \longrightarrow \varepsilon(M[\varepsilon(A, Env)], \langle Id, Empty\rangle)$

where the notation M[A] stands for the λ_{fc} term obtained by naively substituting A for all occurrences of i in M that are bound by the abstraction $\lambda.M$.

Let ν be the occurrence of the graph rule that is reduced in the reduction $G \longrightarrow G'$. Let μ be the approximating morphism from T to G. Let

$$N_{ap} = \{m \mid m \in \mu^{-1}(\nu(n)) \text{ and } lab(m) \neq \square\}$$
$$N_{\lambda} = \{m \mid m \in \mu^{-1}(\nu(n_2)) \text{ and } lab(m) \neq \square\}$$
$$N_{i} = \{m \mid m \in \mu^{-1}(\nu(n_1)) \text{ and } lab(m) \neq \square\}$$

The set N_{ap} is the set of nodes T that maps to the root of the redex in G. Note that N_{ap} may be empty. We will assume that N is non-empty, which is the general case.

Part 1. Consider a node in the set N_{ap}. Proceeding as in the proof of Lemma 6.36, we show that its label must be *ap*. Moreover, its first successor must have label λ, since $\square$ nodes cannot be first successor of application nodes. Thus, the subtree rooted at a node in N_{ap} must be an instance of the LHS of the (β_{fc}) rule for terms. Now, a β_{fc} redex in a term never interferes with another β_{fc} redex for the following reasons. The pattern part of a β_{fc} redex in a term consists of the *ap* node, the λ node, and all the i nodes that are bound by the λ node. Now, the *ap* and the λ node of a β_{fc} redex cannot be part of the pattern of another β_{fc} redex. Moreover, Proposition 6.10 says that an i node can be bound by at most one λ-abstraction.

Part 2. The proof is a little difficult as well as tedious for the following reasons. First, we have to handle redirection of pointers for two nodes. Second, the redexes reduced in going from T to T' are not disjoint. So an inner redex may contain i nodes that are bound by an outer redex. The substitution for these i nodes takes place only when the outer redex is reduced. On the other hand, the application of the graph rule to G redirects, in one step, all pointers to the i node to the argument node of the redex. Thus, the reduct of the inner redex in T does not approximate a subgraph of the graph after redirection of pointers, as was the case in Lemma 6.36.

To shorten the proof, we will use the following notation. $\vec{a}$ denotes a sequence of objects; the length of the sequence and objects in the sequence will be clear from the context. $T[\vec{m} := \vec{P}]$ denotes either T or $T[m_1 := P_1, \ldots m_k := P_k]$ depending upon whether $\vec{n}$ is empty or not.

It will also be useful to rephrase the β_{fc} rule for trees. Suppose $(\lambda.M)\ A$ is a β_{fc} redex, and let $\vec{b}$ be the sequence of all i nodes that are bound by the abstraction part of the redex. Let $S \equiv \varepsilon(A, Env)$, and let $\vec{S}$ be a sequence containing as many copies of S as the length of $\vec{b}$. Then, the reduct of the redex is

$\varepsilon(M[\vec{b} := \vec{S}], \langle Id, Empty\rangle)$

First, we show that N_i contains all and only those i nodes of T that are substituted in the reduction from T to T'. Since G represents a De Bruijn term, the second condition in Definition 6.30 implies that N_i includes all and only those i nodes that are bound by members of N_λ. Now, each node in N_λ represents the abstraction part of a redex rooted at a node in N_{ap}. This follows from the condition attached to the rule, which implies that the node $\nu(n)$ of G, *i.e.*, the *ap* node of the redex, is the only node that has pointer to $\nu(n_2)$, *i.e.*, the λ node.

The graph G' is given by

$$\begin{array}{rcl} B & = & G +_\nu (G, n, n', n_1, n_1') \\ C & = & B[\nu(n) := n'][\nu(n_1) := n_1'] \\ G' & = & GC(C). \end{array}$$

Actually, the order of redirecting pointers for two nodes doesn't matter. But we will assume that the redirection is done in the indicated order.

Now, we show that T' approximates G'. To keep track of i nodes of T that are in the set N_i, we underline all nodes of T that are in N_i and carry the underlining through the reduction in the obvious way. Then, we claim the following.

Claim: Let T_1 be a subtree of T, and suppose μ maps the root of T_1 to the node m of G. Suppose $T_1 \twoheadrightarrow T_1'$ by an innermost complete reduction relative to redexes whose roots are in N_{ap}. Let εu be a sequence of all underlined i nodes of T_1'. Let $\vec{P}$ be a sequence of trees such the its length is the same as that of $\vec{n}$. Moreover, assume that each tree in $\vec{P}$ approximates $C|n_1'$. Then, $T_1'[\vec{u} := \vec{P}]$ approximates $C|\phi(m)$ where

$$\phi(m) = \begin{cases} n', & \text{if } m = \nu(n) \\ n_1', & \text{if } m = \nu(n_1) \\ m, & \text{otherwise} \end{cases}$$

The proof is by induction on the number of nodes of T_1 that belongs to the set N_{ap}.

Base Step. Suppose T_1 contains no nodes that are in the set N. Then, $T_1' \equiv T_1$. Let $C_1 = B[\nu(n) := n']$. So $C = C_1[\nu(n_1) := n_1']$. Proceeding as in Part 2(a) of Lemma 6.33, we show that T_1 also approximates $C_1|m$. Now, let $\vec{n}$ and $\vec{P}$ be as in the statement of the claim. Then, Proposition 6.27 implies the following. $T[\vec{n} := \vec{P}]$ approximates $C|n_1'$, if $m = \nu(n_1)$; otherwise, it approximates $C|m$. By the definition of $\phi(m)$, we have that $T[\vec{n} := \vec{P}]$ approximates $C|\phi(m)$.

Induction step. Suppose T_1 contains k nodes that are in the set N.

First, suppose the root of T_1 is in N_{ap}. Thus, $m = \nu(n)$ and $\phi(m) = n'$. Since redexes rooted at the nodes in N don't interfere, the reduction from T_1 to T_1' is of the form

$$\begin{aligned} T_1 &\equiv (\lambda.M)\, A \\ &\longrightarrow (\lambda.M')\, A' \\ &\longrightarrow \varepsilon(M'[\vec{b} := \vec{S}]) \end{aligned}$$

where M' is obtained from M by an innermost complete reduction relative to the redexes whose roots are in N_{ap}, A' is obtained from A similarly, $\vec{b}$ is a sequence of all i nodes that are bound by $\lambda.M'$, and $\vec{S}$ is a sequence containing as many copies of $S \equiv \varepsilon(A', Env)$ as the length of $\vec{b}$.

Now, we claim that nodes in $\vec{b}$ must be underlined i nodes. The underlined i nodes of $\lambda.M'$ are traces of underlined i nodes of $\lambda.M$, and underlined i nodes of $\lambda.M$ include i nodes bound by the λ-abstraction. The claim follows from Proposition 6.9 and the result that β_{fc} reduction correctly simulates β reduction in De Bruijn notation.

Since T_1 approximates $G|\nu(n)$, Proposition 6.24 and properties of μ and ν imply that A approximates $G|\nu(o_2)$. Also, the number of nodes of A that belong to the set N_{ap} is less than k. So let $\vec{a}$ be a sequence of all underlined i nodes of A', and let $\vec{P}$ be a matching sequence of trees, each of which approximates $C|n_1'$. By induction hypothesis, $A'[\vec{a} := \vec{P}]$ approximates $C|\phi(\nu(o_1))$. By the definition of the operation of redirecting pointers, $\phi(\nu(o_1))$ is simply the first successor of the node n_1' in C. Then it is easy to see that $S[\vec{a} := \vec{P}]$ approximates $C|n_1'$. (Note that we are using the same names for underlined i nodes of A' and $S \equiv \varepsilon(A', Env)$. This shouldn't be confusing.)

Similarly, we show the following. Let $\vec{f}$ be a sequence of all underlined i nodes of M', and let $\vec{Q}$ be a matching sequence of trees, each of which approximates $C|n_1'$. Then, $M'[\vec{f} := \vec{Q}]$ approximates $C|n_s'$ where n_s' is the first successor of n'.

Now, node in $\vec{f}$ includes all nodes in $\vec{b}$. Also, since $S[\vec{a} := \vec{P}]$ approximates $C|n_1'$, we can replace a tree in $\vec{Q}$ by this tree. So let $\vec{f_-}$ be all those nodes of $\vec{f}$ that are not in $\vec{b}$, and let $\vec{Q_-}$ be the sequence of trees that corresponds to nodes in $\vec{f_-}$. Then,

$$M'[\vec{b} := \vec{S}[\vec{a} := \vec{P}], \vec{f_-} := \vec{Q_-}]$$

approximates $C|n_s'$. Instead of replacing underlined i nodes in S first, we may first replace nodes in $\vec{b}$ by S and then replace underlined i nodes in each copy of S. Suppose $\vec{a_j}$ is a sequence of underlined i nodes in the jth copy of S. Then,

$$(M'[\vec{b} := \vec{S}])[\vec{a_1} := \vec{P}], \ldots, \vec{a_j} := \vec{P}], \ldots, \vec{f_-} := \vec{Q_-}]$$

approximates $C|n'_s$. Then, it is easy to see that

$$(\varepsilon(M'[\vec{b} := \vec{S}], \langle Id, Empty \rangle))[\vec{a_1} := \vec{P}], \ldots, \vec{f_-} := \vec{Q_-}]$$

approximates $C|n'$. The claim then follows from the fact that nodes in $\vec{a_1}, \ldots, \vec{f_-}$ are all the underlined nodes in the reduct of the redex.

Now, suppose the root of T_1 is not in the set N_{ap}. Then, $\phi(m) = m$. Consider all those redexes in T_1 whose roots are in the set N_{ap} and among them consider the ones that are outermost. Let $R_1, \ldots R_j$ be these redexes, and let $m_1, \ldots m_j$ be their roots. Then,

$$T_1' \equiv T_1[m_1 := R_1', \ldots, m_j := R_j']$$

and, for $1 \leq i \leq j$, $R_i \twoheadrightarrow R_i'$ by an innermost complete reduction relative to the redexes whose roots are in N. Let $\vec{u_0}$ be underlined i nodes of T that are not in $R_1, \ldots, R_j$, and let $\vec{u_i}$ be a sequence of all underlined i nodes in R_i'. We show that

$$T_1'' = T_1[\vec{u_0} := \vec{P_0}, m_1 := R_1[\vec{u_1} := \vec{P_1}], \ldots]$$

approximates $C|m$ where $\vec{P_0}, \ldots, \vec{P_j}$ are as in the statement of the claim. Since T_1 approximates $G|m$, it also approximates $B|m$ (see Part 2(a) of Lemma 6.33). As shown above $R_j'[\vec{u_j} := \vec{P_j}]$ approximates $C|n'$. Given these results, we prove that T_1'' approximates $C|m$ in a way similar to the proof of Proposition 6.27. Note that Proposition 6.27 doesn't apply directly as there are two redirections involved in going from B to C. Now,

$$T_1'' = T_1'[\vec{u_0} := \vec{P_0}, \ldots, \vec{u_j} := \vec{P_j}].$$

The claim then follows from the fact that $\vec{u_0}, \ldots, \vec{u_j}$ are all underlined i nodes in T_1'.

This completes the proof of the claim.

Now, we complete the proof of Part (2). There are no underlined i nodes in T', since all underlined i nodes get substituted in the reduction of T to T'. Thus, the claim implies that T' approximates $C|\phi(r_G)$. But $\phi(r_G) = \phi(r_B) = r_C$. So T' approximates $C|r_C = GC(C) = G'$.

□

Lemma 6.38 *Same as Lemma 6.36 but for (λ-sp). In this case the reduction of $LFC(G)$ to T' uses both (λ-spL) and (λ-spR).*

Proof. Proof of Part (1) is obvious. A node that is mapped to the λ node of the redex in G and is not labelled □ must be labelled with λ. Hence it is the root of a (λ-spL) or (λ-spR) redex.

The result that these redexes don't interfere follows from Proposition 6.10, which states that an i node can be bound by at most one λ-abstraction.

Now, we consider Part (2). Recall that in applying the graph rule, all pointers from a distinguished node, $\nu(m)$ where ν is the occurrence of the rule, are redirected to one copy of the λ node, and all other pointers are redirected to another copy of the λ node. So we divide the λ nodes that are mapped to the λ node of the redex in G into two classes: ones that are pointed at by the nodes that are mapped to $\nu(m)$, and all others. In reducing T to T', we use the rule (λ-spL) for the λ nodes in the first class and use (λ-spR) for the λ nodes in the other class. The rest of the proof proceeds as for the β_{fc} rule. (Actually, it is simpler than for the β_{fc} rule.) □

Now, we can give the proof of Theorem 6.32

Proof of Theorem 6.32:

Part 1. We have to show that G' satisfies the conditions of Definition 6.30. We show that G' satisfies the first condition. By Lemma 6.35, 6.36, 6.37 or 6.38, $LFC(G) \underset{E_c,F_c}{\twoheadrightarrow} T'$ and T' approximates G'. By Theorem 6.11, T' represents a De Bruijn term. The result follows.

Now, we show that G' satisfies the second condition. The result follows form the following observations. First, n has a binding pointer to n' in G' iff n has a binding pointer to n' in G. Second, an i node in $LFC(G')$ is bound by a λ-node iff they are traces of i and λ-node in $LFC(G)$ such that the i node is bound by the λ-node. For β_{fc} rule, this follows from the correspondence between λ_{fc} calculus and De Bruijn's notation. The result follows by noting how the traces of a node are mapped onto nodes in G'.

Part 2. Immediately follows from Part 1 and Lemmas 6.35, 6.36, 6.37 and 6.38. □

6.4.2 Correctness of the interpreter

Using Theorem 6.32 and the correspondence between λ_{fc} calculus and De Bruijn notation, we have the following.

Theorem 6.39 *Let G be a λ_{fc} graph that represents a De Bruijn term. Then, the following two diagrams commute.*

Proof. Immediate from Theorem 6.32 and Theorem 6.11.

Theorem 6.40 *Let M be a λ-expression in De Bruijn notation and, let G be the initial graph representation of M. Then, the reduction of G by the interpreter terminates iff M has a normal form. If the reduction terminates, then the final graph translates to the normal form of M. Moreover, the translation can be done by simply unraveling the final graph and replacing each subexpression of the form $\varepsilon(i, Snd \circ Fst^n)$ by n.*

Proof. We show that if $G \twoheadrightarrow G'$ where the redex is chosen according the strategy described in Section 4.4.5, then the reduction of $LFC(G)$ to $LFC(G')$ is the reduction of a non-empty set of redexes. Also, we show that if $G \underset{\beta_{fc}}{\twoheadrightarrow} G'$ where the redex is chosen according the strategy described in Section 4.4.5, then the reduction $Tr(G) \twoheadrightarrow Tr(G')$ reduces the leftmost reduction in $Tr(G)$. These two results establish that the reduction of G terminates iff M has a normal form.

Now, suppose the reduction of G terminates in G'. Since G' is in normal form, $LFC(G')$ is also in normal form. Since $LFC(G')$ represents a De Bruijn term and is in normal form, it can contain no $\square$ nodes. The claim then follows from 6.25. $\square$

6.5 Proof of Optimality

In this section, we show that the interpreter correctly implements Lévy's theoretical specification of optimal reductions in the λ-calculus.

To relate reductions in the labelled λ-calculus and reductions on λ_{fc} graphs performed by the interpreter, we use the λ_{fc} calculus as an intermediate point. As discussed later, labelling of subexpressions in labelled λ-terms can be transferred to labelling of applications, λ-abstractions and i's in λ_{fc} terms in an obvious way.

In this section, we will be concerned with only those λ_{fc} graphs and λ_{fc} terms that represent De Bruijn terms. So we will drop the qualification and refer to them simply as λ_{fc} graphs and λ_{fc} terms. Also, we will ignore the distinction between λ-terms in De Bruijn notation and the

usual λ-terms and work only with the latter. As in the last section, we will consider terms as trees.

As mentioned in the introduction, the proof uses a relation, called *l-shared*, on the subexpressions of a labelled expression. Two redexes in a labelled expression are l-shared iff they have the same degree and hence must be reduced together according to Lévy's theory. This forms the base case for the inductive definition given below. For other types of subexpressions, the relation ensures that only subexpressions that are l-shared contribute to the creation of l-shared redexes. In other words, subexpressions that are not l-shared cannot become part of l-shared redexes. The following definition will be useful in defining the relation.

Definition 6.41 An application inside a λ-term M can *become a redex* if there is a reduction $M \twoheadrightarrow N$ such that a descendant of the application (relative to the reduction) is a redex.

Definition 6.42 The relation *l-shared* on sub-expressions of a labelled λ-term is defined as follows. Two subexpressions are l-shared iff one of the following holds.

- Both are applications satisfying the following conditions:
    1. They can become redexes.
    2. Their function parts have identical labels.
    3. Function parts of both applications either are λ-abstractions, or are l-shared applications, or are l-shared variables.
- Both are λ-abstractions satisfying the following conditions:
    1. They have identical labels.
    2. Both either are function parts of applications, or are argument parts of l-shared applications, or are bodies of l-shared λ-abstractions.
- Both are variables satisfying the following conditions: They have the same label, and they are bound by l-shared λ-abstractions.

A few remarks about the definition. First, various conditions in a clause are not necessarily independent; the above phrasing for defining the relation was chosen primarily because it is easier to understand. Second, if two redexes have the same degree, then they as well as their abstraction parts are l-shared. Third, a subtle point implied by the clause for λ-abstractions

is that if an expression is a λ-abstraction, then the abstraction cannot be l-shared with an abstraction inside the expression. Finally, a free occurrence of a variable is never l-shared with another occurrence (free or bound) of a variable.

Proposition 6.43 *An application inside a λ-expression can become a redex iff one of the following holds:*

- *The application is of the form $(\cdots((\lambda x.M)\, N_1)\cdots N_i)$.*
- *The application is of the form $(\cdots(x\, N_1)\cdots N_i)$, x is bound by a λ-abstraction, and the binding λ-abstraction itself is part of an application that can become a redex.*

Proof. Obvious by looking at the way redexes are created.

Proposition 6.44 *Let M be a labelled λ-term, and let $M \twoheadrightarrow N$. If two subexpressions of N are l-shared, then their ancestors in M (relative to the reduction) are l-shared.*

Proof. The proposition follows from the properties of labelled λ-calculus. A detailed proof will be supplied. □

As pointed out earlier, we use λ_{fc} calculus as an intermediate point to relate the reductions performed by the interpreter and reductions in the labelled calculus. Since ε and *cond* nodes have no counterpart in the λ-calculus, we define the following notion.

Definition 6.45 Let T be a λ_{fc} term. An *s-edge* is a non-empty path in T such that the end nodes of the path are *ap*, λ, or i nodes and the intermediate nodes are ε or *cond* nodes. We will write an s-edge as (m, n) where m and n are the end nodes of the path.

With this definition, an intuitive way to think of translation of λ_{fc} terms to λ-terms is that the translation replaces each s-edge by an edge and replaces i's by numbers (or variables) in a way so that binding relationship is preserved.

With this in mind, we can transfer labels of subexpressions in labelled λ-terms to labels of applications, λ-abstractions and i's in λ_{fc} terms. There is, however, a possibility of confusing Lévy's labels attached to subexpressions of λ_{fc} terms and the labelling of nodes by function symbols. In the rest of this section, label of a subexpression (or a node) means Lévy's label attached to the subexpressions; for the other labelling, we will use the phrase "function symbol of a node".

Notation: Let T be a λ_{fc} term that represents λ-term M, and let M_L be a labelled version of M. By the label of an application, λ-abstraction, or i in T, we mean the following. In T, consider each s-edge as simply an edge, then transfer the labels of M_L to nodes in T in an obvious way with the label of an i node being the label of the corresponding variables occurrence in M_L. Somewhat more precisely, find the $E_c \cup F_c$ normal form of T, say T'. Transfer labelling of M_L to T' in the obvious way. Then, transfer the labelling of T' to T by making the label of a subexpression in T to be the same as the label of its trace in T'. We will write *label*(n) for the label of a node n of T.

Given this, we can also transfer the relation l-shared to subexpressions of λ_{fc} terms.

The following definition captures sharing in λ_{fc} graphs.

Definition 6.46

1. Two s-edges in a λ_{fc} tree are *similar* iff they they are of the form

$$(n_1, i_1, \ldots, i_{k-1}, n_k)$$
$$(n'_1, i_1, \ldots, i_{k-1}, n'_k),$$

and, for $1 \leq j \leq k$, $lab(n_j) = lab(n'_j)$.

2. Let G be a λ_{fc} graph, and let μ be the approximating morphism from $LFC(G)$ to G. Let (n_1, n_2) and (n'_1, n'_2) be two s-edges in $LFC(G)$. The two s-edges *map onto the same path* in G iff they are similar and $\mu(n_1) = \mu(n'_1)$.

A few remarks about the last definition. Consider two nodes that occur in the same position in the two s-edges. They must be mapped to the same node in G by μ. If they are ε nodes, then their function parts are identical. If they are *cond* nodes, then their function parts are identical, and they have $\Box$ on the same side. Also note that some of the conditions for being similar are implied by the requirement that n_1 and n'_1 are mapped onto the same node of G.

Now, we can state and prove the main theorem.

Theorem 6.47 *Let M be a λ-term that has a normal form. Let M_0 be a labelled λ-term obtained by labelling every subexpression of M by distinct elementary labels, and let G_0 be the initial graph representation of M. Suppose $G_0 \longrightarrow G_1, \ldots, \longrightarrow G_n$ is the reduction of G_0 to normal form using the interpreter. Let $M_0, \twoheadrightarrow M_1, \ldots, \twoheadrightarrow M_n$ be the labelled version of the reduction $Tr(G_0) \longrightarrow Tr(G_1), \ldots, \longrightarrow Tr(G_n)$. Then, for $0 \leq j < n$, the following holds.*

1. *If $G_j \underset{\beta_{fc}}{\longrightarrow} G_{j+1}$, then $M_i \overset{\mathbf{R_j}}{\longrightarrow} M_{j+1}$ where $\mathbf{R_j}$ contains all and only those redexes that have the same degree and includes leftmost redex in M_j.*

2. If $G_j \underset{\neg\beta_{fc}}{\longrightarrow} G_{j+1}$, then $M_j = M_{j+1}$

Thus, the reduction performed by the interpreter implements leftmost complete reduction beginning with M_0, and the number of β_{fc} redexes reduced by the interpreter is equal to the length of the leftmost reduction.

Proof. All parts except that $\mathbf{R}_j$ includes all and only those redexes in M_j that have the same degree follow from the proof of correctness. To prove the remaining, we have to relate the sharing present in G_j and labels of subexpressions in M_j. We will show the following.

Claim: For $0 \leq j \leq n$, the following holds.

(1) The set of redexes reduced in any non-empty step of $M_0 \twoheadrightarrow, \ldots, M_j$ includes all and only those redexes that have the same degree.

(2) The structure of G_j and labels of subexpression in M_j are related a follows. Transfer the labels of subexpression of M_j to $LFC(G_j)$. Let (p, q) and (p', q') be two s-edges in $LFC(G_j)$. Then,

C1. If the two s-edges map onto the same path in G, then q and q' have the same label.

C2. If all the following conditions hold, then the two s-edges map on the same path in G: p and p' are l-shared, q and q' are l-shared, and q and q' have the same label.

The claim immediately implies the result. The proof of the claim is by induction on j.

Base case: We only have to prove the second part. Since G_0 is the initial graph representation of a λ-term, the only nodes that are shared are i nodes. Thus, no two s-edges of $LFC(G_0)$ map onto the same path in G_0, and (C1) is vacuously true.

Since all subexpressions of M_0 are labelled with distinct labels, no two s-edges of $LFC(G_0)$ satisfy the conditions for applying (C2). Again, (C2) is vacuously true.

Induction step: Suppose the claim is true for $j < n$. We prove that the claim is true for $j + 1$.

Proof of Part 1: If $G_j \underset{\neg\beta_{fc}}{\longrightarrow} G_{j+1}$, then then $M_j \equiv M_{j+1}$, and the result follows from the induction hypothesis. If $G_j \underset{\beta_{fc}}{\longrightarrow} G_{j+1}$, then $M_j \xrightarrow{\mathbf{R}_j}\!\!\!\!\twoheadrightarrow M_{j+1}$. We show that $\mathbf{R}_j$ includes all and only those redexes of M_j, which along with the induction hypothesis implies the result.

It follows from the proof of correctness that the set $\mathbf{R}_i$ contains redexes that are traces of redexes reduced in the reduction $LFC(G_j) \twoheadrightarrow LFC(G_{j+1})$. By Lemma 6.37, the redexes reduced in the last reduction are the ones whose roots are in the set

$$N_{ap} = \{m \mid m \in \mu^{-1}(n) \text{ and } lab(m) \neq \Box\}$$

where μ is the approximating morphism from $LFC(G_j)$ to G_j and n is the root of the redex in G_j.

First, we show that redexes in $\mathbf{R}_j$ have the same degree. Note that the set $\mathbf{R}_j$ is not empty. If it contains only one redex, then there is nothing to prove. So consider two redexes in the set, and consider the s-edges from the *ap* nodes to the λ nodes of the corresponding redexes in $LFC(G_j)$. These s-edges are of the form $(p, 1, q)$ and $(p', 1, q')$ where p', q' are *ap* nodes and q, q' are λ nodes. Thus, they are similar. Also, p and p' belong to the set N_{ap}. Thus, the two s-edges map onto the same path in G_j. By (C1), the q and q' have the same label. The result follows.

Now, we show that $\mathbf{R}_j$ includes all redexes of M_j that have the same degree. The proof is by contradiction. Suppose M_j contains a redex that have the same degree as a redex in the set $\mathbf{R}_j$ but is not in the set. Consider the s-edge, say (p, q), from the *ap* node to the λ node of the corresponding redex in $LFC(G_j)$. Then, the node p is not is the set N_{ap}. Now, take a redex in $\mathbf{R}_j$, and consider the s-edge, (p', q') from the *ap* node to the λ node of the corresponding redex in $LFC(G_j)$. Since, the two redexes have the same degree, conditions for applying (C2) follow trivially. Thus, s-edges (p, q) and (p', q') must map on the same path in G_j. But then p belongs to N_{ap}. Contradiction.

Proof of Part 2: This is, of course, the main part of the proof. The proof is by cases on the rule applied in the reduction of G_j to G_{j+1}. To simplify writing the proof, we assume the following.

1. $G \equiv G_i$, $T \equiv LFC(G)$, and μ is the approximating morphism from T to G.

2. $G' \equiv G_{i+1}$, $T' \equiv LFC(G')$, and μ' is the approximating morphism from T' to G'.

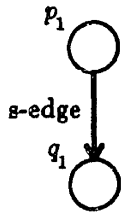
3. (p'_1, q'_1) and (p'_2, q'_2) are two s-edges in T'.

4. p_1, q_1, p_2, and q_2 are nodes of T such that

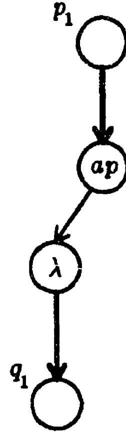
$$p_1 = trace^{-1}(p'_1) \quad q_1 = trace^{-1}(q1')$$
$$p_2 = trace^{-1}(p'_2) \quad q_2 = trace^{-1}(q2')$$

Note that $T \longrightarrow T'$ by an innermost reduction of a set of redexes in T', and μ maps the roots of these redexes to the root of the redex reduced in the reduction of G to G' (see Theorem 6.32 and proof of correctness).

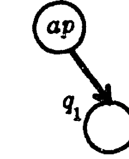
β_{fc}**rule:** Since (p_1', q_1') is an s-edge in T', we have the following four cases concerning nodes p_1, q_1 in T:

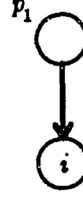


(1)

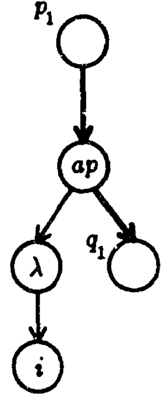


(2)





(3)



(4)

In cases other than (1), the displayed *ap* node is the root of a redex that is reduced in the reduction from T to T'. In the last two cases, the displayed i node is bound by the abstraction part of the the displayed *ap* node. Similarly, we have the four cases concerning nodes p_2, q_2. As we shall see, nodes p_1, q_1 and nodes p_2, q_2 must obey the same case.

Also, note that the construction of μ' from μ is such that, for any two nodes s, s' of T' that are labelled with function symbols *ap*, or λ or i,

$$\mu'(s) = \mu'(s') \text{ iff } \mu(trace^{-1}(()s)) = \mu(trace^{-1}(()s')).$$

See the proof of Lemma 6.37.

Proof of C1: Suppose (p_1', q_1') and (p_2', q_2') map onto the same path in G'. We have to show that q_1 and q_2 have the same label.

First, we show that p_1, q_1 and p_2, q_2 must obey the same case. Note that the reduction of G to G' introduces two new ε nodes, one at the root of the contractum and one at the place of i node. Also, since (p_1', q_1') and (p_2', q_2') map onto the same path in G', nodes in the same position in two s-edges must be mapped on the same node of G'. Now, we consider the following the three cases.

Suppose p_1, q_1 obey the first case. Then, p_2, q_2 cannot be in cases (2), (3), or (4) because then (p_2', q_2') includes a node that is mapped to one of the new ε nodes but (p_1', q_1') doesn't.

Suppose p_1, q_1 obey the second case. Then, p_2, q_2 cannot be in cases (3), or (4) because then (p_2', q_2') includes a node that is mapped to the new ε node introduced at the place of i but (p_1', q_1') doesn't.

Suppose p_1, q_1 obey the third case. Then, p_2, q_2 cannot be in case (4) because then (p_2', q_2') includes a node that is mapped to the new ε node at the root of the contractum but (p_1', q_1') doesn't.

The above along with the symmetry of cases imply that p_1, q_1 and p_2, q_2 must obey the same case.

Thus, to prove C1, we consider the following four cases:

1. p_1, q_1 and p_2, q_2 obey the first case. Then, (p_1, q_1) and (p_2, q_2) are similar as they are identical to (p_1', q_1') and (p_2', q_2'), respectively. Also, $\mu(p_1) = \mu(p_2)$. Thus, by the induction hypothesis, q_1 and q_2 have the same label. But labels of q_1' and q_2' are identical to the labels of q_1 and q_2, respectively. The result follows.

2. p_1, q_1 and p_2, q_2 obey the second case. Let ap_1 and ap_2 be the corresponding application nodes, and λ_1 and λ_2 be the corresponding λ nodes. Now, (p_1', q_1') is obtained by replacing ap-λcombination in the path from p_1 to p_2 by an ε node. Similar comments apply for (p_2', q_2'). Since (p_1', q_1') and (p_2', q_2') are similar, (p_1, ap_1) and (p_2, ap_2) are similar. Since $\mu(p_1) = \mu(p_2)$, the two s-edges (p_1, ap_1) and (p_2, ap_2) map onto the same path in G. By induction hypothesis, ap_1 and ap_2 have the same label, say u. Similarly, we show that q_1 and q_2 have the same label, say w. Since the redexes reduced in the reduction from T to T' have the same degree, λ_1 and λ_2 have the same label, say v. Then, labels of both q_1' and q_2' are $w\overline{v}u$, and the result follows

3. p_1, q_1 and p_2, q_2 obey the third case. Let ap_1 and ap_2 be the corresponding application nodes, and i_1 and i_2 be the corresponding i nodes. Proceeding as in (2), we show that i_1 and i_2 have the same label, say u, and q_1 and q_2 have the same label, say w. Also, the abstraction parts of the redexes have the same label, say v. Then, labels of both q_1' and q_2' are $w\underline{v}u$, and the result follows.

4. p_1, q_1 and p_2, q_2 obey the fourth case. This case is a combination of (2) and (3).

Proof of C2: Suppose the required conditions hold. Then, we have to show that (p_1', q_1') and (p_2', q_2') map onto the same path in G'.

The following observation will be used heavily in the proof. Let v be the degree of redexes reduced in the reduction of T to T'. Then, no label in T can contain $\underline{v}$ or $\overline{v}$. This follows from Proposition 2.12 and the inductive assumption that the reduction of M_0 to M_i is a complete

reduction beginning with an elementarily labelled term. Note that labels of nodes in T are labels of subexpressions in M_i.

Now, we show that p_1, q_1 and p_2, q_2 must obey the same case by showing that if they obey different cases then q_1' and q_2' cannot have the same label, which is one of the assumption.

Suppose p_1, q_1 obey the first case. Then, p_2, q_2 cannot be in cases (2), (3), or (4) because the label q_2' contains $\underline{v}, \overline{v}$, or both, but the label of q_1', which is the same as label of q_1, cannot (see above).

Suppose p_1, q_1 obey the second case. Then, p_2, q_2 cannot be in cases (3), or (4) because then the label of q_2' contains $\underline{v}$ but the label of q_1' cannot.

Suppose p_1, q_1 obey the third case. Then, p_2, q_2 cannot be in case (4) because then the label of q_2' includes $\overline{v}$ but the label of q_1' cannot.

The above along with the symmetry of cases imply that p_1, q_1 and p_2, q_2 must obey the same case.

Before we proceed, note that Proposition 6.44 implies that p_1, p_2 are l-shared and q_1, q_2 are l-shared. (Also using the fact that the definition of traces for β_{fc} reduction is identical to the definition of descendants.) Now, we consider the following four cases:

1. p_1, q_1 and p_2, q_2 obey the first case. Then, q_1 and q_2 have the same label. By induction hypothesis, (p_1, q_1) and (p_2, q_2) map onto the same path in G. Thus, they are similar and $mu(p_1) = \mu(p_2)$. Then, (p_1', q_1') and (p_2', q_2') are similar, and as noted at the beginning of the proof for this rule, $\mu(p_1') = \mu(p_2')$. The result follows.

2. p_1, q_1 and p_2, q_2 obey the second case. Let ap_1 and ap_2 be the corresponding application nodes; the two are, of course, l-shared. Let λ_1 and λ_2 be the corresponding λ nodes; again the two are l-shared. Now, ap_1, ap_2 must have the same label, and q_1, q_2 must have the same label. For if that is not the case, then q_1', q_2' have the same label implies that a label in T includes $\underline{v}$ or $\overline{v}$, which is impossible. By induction hypothesis, (p_1, ap_1) and (p_2, ap_2) map onto the same path in G. Similarly, (λ_1, q_1) and (λ_2, q_2) map onto the same path in G. Then, it is easy to see that (p_1', q_1') and (p_2', q_2') map onto the same path in G'.

3. p_1, q_1 and p_2, q_2 obey the third case. Let ap_1 and ap_2 be the corresponding application nodes; the two are l-shared. Let i_1 and i_2 be the corresponding i nodes; again, the two are l-shared. Proceeding as in (2), we show that (p_1, i_1) and (p_2, i_2) map onto the same path in G and that (ap_1, q_1) and (ap_2, q_2) map onto the same path in G. Then, it is easy

to see that (p_1', q_1') and (p_2', q_2') map onto the same path in G'.

4. p_1, q_1 and p_2, q_2 obey the fourth case. This case is a combination of (2) and (3).

This completes the proof for the β_{fc} rule.

Proof for the other rules: We make some observations that are common to all rules. Since both T and T' represent the same λ-term, (p_1, q_1) and (p_2, q_2) are s-edges of T. Also, since labelling of nodes for the two are derived from the same labelled term, we have that for any node n of T' that is labelled with function symbol ap, λ, or i,

$$label(n) = label(trace^{-1}(()n))$$

Then, to prove (C1), we simply have to show that (p_1', q_1') and (p_2', q_2') map onto the same path in G' implies that (p_1, q_1) and (p_2, q_2) maps onto the same path in G. The result follows from inductive assumption and the above observation about labels.

Now, suppose we want to prove (C2). Then, the observation about labels and assumptions in (C2) imply the following: p_1 and p_2 are l-shared, q_1 and q_2 are l-shared, and q_1 and q_2 have the same label. Then, by induction hypothesis, (p_1, q_1) and (p_2, q_2) maps onto the same path in G. To prove (C2), then, we have to show that under these conditions, (p_1', q_1') and (p_2', q_2') map onto the same path in G'.

λ-sp Rule: Recall that in applying the graph rule, all pointers from a distinguished node, say m, are redirected to one copy of the λ node, and all other pointers are redirected to another copy of the λ node. Also, as noted in Lemma 6.38, the reduction of T to T' is done by applying (λ-spL) and (λ-spR); the first rule is applied to all λ nodes that has pointers from nodes that are mapped to m, the second rule is applied to all other λ nodes.

We consider the following four cases based on the structure of s-edges (p_1, q_1) and (p_2, q_2).

1. The edges don't contain λ or i nodes that are involved in the reduction of T to T'.

2. q_1 and q_2 are λ nodes involved in the reduction.

3. p_1 and p_2 are λ nodes involved in the reduction.

4. q_1 and q_2 are i nodes involved in the reduction.

No other case can arise. In the proof of (C1), this follows from the assumption that (p_1', q_1') and (p_2', q_2') map to the same path in G'. In the proof of (C2), this follows from the result that (p_1, q_1) and (p_2, q_2) map to the same path in G (see the general comment).

Proof of C1: Suppose (p'_1, q'_1) and (p'_2, q'_2) map to the same path in G'.

Cases 1 and 2. Obvious as neither the structure of s-edges nor the mapping for their source nodes are affected by reduction (see the similar case for the β_{fc} rule).

Case 3. The only difference between s-edge (p_1, q_1) and (p'_1, q'_1) is that the latter edge contain an ε node after the λ node. Similar comments apply for (p_2, q_2) and (p'_2, q'_2). Since (p'_1, q'_1) and (p'_2, q'_2) are similar (p_1, q_1) and (p_2, q_2) are similar. Moreover, $\mu(p_1) = \mu(p_2)$ because both are λ nodes involved in the reduction. The result follows.

Case 4. This case is similar to Case 3. The result that $\mu(p_1) = \mu(p_2)$ is implied by assumption that $\mu'(p'_1) = \mu(p'_2)$.

Proof of C2: We have that (p_1, q_1) and (p_2, q_2) map to the same path in G.

Cases 1 and 2. Again obvious.

Case 3. This is the important case, for it shows that duplicating λ nodes as done by this rule preserves optimality. First, we claim that a λ node to which the (λ-spL) rule is applied cannot be l-shared with a λ-node to which the (λ-spR) rule is applied. Suppose they are l-shared. The definition of l-shared implies that the λ nodes must have the same labels, and that sources of s-edges pointing to λ nodes must be l-shared. Thus, all the conditions for applying (C2) to s-edges that are pointing to λ nodes are satisfied. Hence, the two s-edges must map on the same path in G. But that is impossible as one of these s-edges includes a node that is mapped to the distinguished node, m, and the other doesn't.

Since p_0 and p_2 are l-shared, they must be λ nodes for which the same rule is applied. Then, the rest is quite easy.

Case 4. The argument given in (2) and the fact that two i nodes are l-shared only if their binding λ-abstractions are l-shared implies the following: *cond* nodes substituted for q_1 and q_2 have $\square$ on the same side. The rest is easy.

This completes the proof for the (λ-sp) rule.

Ap-c Rule: We consider the following cases based on the structure of s-edges (p_1, q_1) and (p_2, q_2).

1. The edges don't contain the *ap* nodes involved in the reduction of T to T'.

2. p_1 and p_2 are *ap* nodes involved in the reduction, and q_1 and q_2 are their function parts.

3. p_1 and p_2 are *ap* nodes involved in the reduction, and q_1 and q_2 are their argument parts.

4. q_1 and q_2 are *ap* nodes involved in the reduction.

No other case can arise. In the proof of (C1), this follows from the assumption that (p'_1, q'_1) and (p'_2, q'_2) map to the same path in G'. In the proof of (C2), this follows from the result that (p_1, q_1) and (p_2, q_2) map to the same path in G (see the general comment).

The proof of (C1) for all cases is quite easy. In proving (C2), the important and non-trivial cases are (3) and (4). They show that duplicating *ap* node doesn't destroy optimality.

First, we show that an *ap* node to which the (Ap-cL) rule is applied cannot be l-shared with an *ap* node to which the (Ap-cR) rule is applied. Suppose they are l-shared. The definition of l-shared implies that the function parts of these *ap* nodes must have the same label and must be l-shared. So consider the s-edges from the *ap* nodes to their function parts. Now, all the conditions for applying (C2) to these s-edges Hence, the two s-edges must map on the same path in G, and hence must be similar. But that is impossible because *cond* nodes immediately following *ap* nodes have □ on different sides.

Since p_1 and p_2 are l-shared, they must be *ap* nodes for which the same rule is applied. Then, (3) and (4) are easy to prove.

ε-ap Rule: We consider the following cases based on the structure of s-edges (p_1, q_1) and (p_2, q_2).

1. The edges don't contain the *ap* nodes involved in the reduction of T to T'.

2. p_1 and p_2 are *ap* nodes involved in the reduction.

3. q_1 and q_2 are *ap* nodes involved in the reduction.

No other case can arise. In the proof of (C1), this follows from the assumption that (p'_1, q'_1) and (p'_2, q'_2) map to the same path in G'. In the proof of (C2), this follows from the result that (p_1, q_1) and (p_2, q_2) map to the same path in G (see the general comment).

The proof of (C1) for all cases is quite easy. In proving (C2), the important cases are (2) and (3); they will show that duplicating *ap* node doesn't destroy optimality. We simply show that these cases don't arise as all *ap* nodes that are involved in the reduction cannot become redexes and hence cannot be l-shared with each other. Since p_1 and p_2 are l-shared, (2) is not possible. Similarly, (3) is not possible.

The reduction strategy for applying rules is such that all nodes in T that are involved in the reduction must be of the form:

$$(\ldots(i, f)\ldots)$$

Such applications can become redexes only if i is bound by a λ-abstraction that is part of a redex. Now, consider the leftmost redex, say R, in T that is reduced in the reduction of T to T'. To the left of this there are no redexes. Now, consider the λ-abstraction, L that binds the displayed i in R. We claim that this λ-abstraction binds displayed i's for *redexes*. Since this λ-abstraction is cannot be part of a redex, none of the *ap* nodes can become redexes. Suppose the claim is not true, and suppose there are two λ-abstractions that binds the i's. Since all i nodes map on to the same i node in G, the definition of λ_{fc} graph that represent De Bruijn terms imply that the two λ-abstractions must map on the same λ node in G. Consider the path to this λ node in G. Then, a node on this path must have two pointers. But this path includes only λ nodes. So there is a λ node with two pointers, that is, an instance of the (λ-sp) rule. But that is not possible.

Remaining rules: Both (C1) and (C2) follow easily. Basically, (p_1, q_1) and (p_2, q_2) are similar iff (p_1', q_1') and (p_2', q_2') are similar, and $\mu(p_1) = \mu(p_2)$ iff $\mu'(p_1') = \mu'(p_2')$.

This completes the proof of the theorem. □

Corollary 6.48 *The interpreter is optimal.*

6.6 I

n this rather long chapter, we showed that the interpreter correctly implements the λ-calculus. We also proved that the interpreter realizes Lévy's specification of an optimal reduction strategy for the λ-calculus.

Chapter 7

Conclusions

The work presented in this thesis was motivated by our desire to design an interpreter for the λ-calculus that implements Lévy's specification of an optimal reduction strategy for the λ-calculus.

The main contribution of this thesis, we believe, is a graph reduction technique that permits us to share *contexts* as opposed to only subexpressions. The idea of sharing contexts can be viewed as a natural extension of the idea of sharing subexpressions, first proposed by Wadsworth in his thesis in 1971 [39]. He developed a graph reduction interpreter for the λ-calculus and gave proof of its correctness. His ideas gained popularity when part of the functional language community started paying serious attention to lazy functional languages. Most interpreters for lazy functional languages now use some mechanism to share subexpressions. Although Wadsworth's interpreter reduced λ-expressions to normal form, most interpreters for lazy functional languages either work with combinators [36, 18] or don't attempt to find normal forms, *e.g.*, environment based interpreters. Our work may have some impact in areas where a λ-calculus interpreter that truly finds normal forms are needed. An example that comes to mind, though we haven't thought about it in detail, is optimizing programs at compile time by partial or symbolic evaluation. Other area where the work may be useful is in designing automatic theorem provers and proof editors.

In Chapters 3 and 4, we developed a graph reduction technique that permits us to share contexts and developed an interpreter for the λ-calculus. We introduced conditional nodes to keep together different subexpressions filling the hole in a shared context and introduced functions on environment-like structures to interpret conditional nodes. In developing the technique, we borrowed ideas from both the literal substitution model and the suspended (or explicit) sub-

stitution model for evaluating λ-expressions. In a way, the notion of sharing contexts tries to combine the best features of the two model. The advantage of the literal substitution model is that substituted expression is immediately available at places where it is needed. The disadvantage, of course, is that some copying must be done. The suspended substitution model has exactly the opposite properties; substituted expression is not made available immediately, but copying is done in a delayed fashion. By sharing contexts, we can make the substituted expression available immediately at places where it is needed and still do copying in a delayed fashion.

In Chapter 4, we presented the interpreter in the form of a graph reduction system alongwith a normalizing strategy for applying the reduction rules. The set of rules includes a version of the β-rule as well as certain other rules, some of which are similar to the rules for handling environments in an environment-based interpreter for the λ-calculus. The interpreter has certain nice features that we would like to mention. First, the input to the interpreter as well as output of the interpreter are "clean" representations of λ-terms; they don't contain conditional nodes. Thus, the translation of the result into a λ-term is quite simple. Second, all the reduction rules used by the interpreter are local constant-time operations; they involve changing a few pointers. Thus, the total number of reductions performed by the interpreter in reducing an expression is a true measure of the cost of reducing the expression. Finally, the reduction strategy is quite simple. We have implemented a version of the interpreter on lisp machines.

To prove the correctness of the interpreter, we developed two calculii, λ_{fc} calculus and λ_f calculus in Chapters 5 and 6. λ_{fc} calculus is essentially the term version of the graph reduction system underlying the interpreter. λ_f calculus is obtained from λ_{fc} calculus by removing certain types of terms, *e.g.*, conditional, and reduction rules that are not very useful for terms. We showed the correspondence between the graph reduction system underlying the interpreter and λ_{fc} calculus as well as correspondence between the two calculii and De Bruijn notation. To relate the graph reduction system to λ_{fc} calculus, we introduced the notion of term approximations to graphs. The notion, though not fully developed in this thesis, may be useful in relating term reduction to graph reduction, especially to reduction of cyclic graphs.

Although λ_f calculus was motivated by the interpreter, it may be of general interest. In the λ_f calculus, changing of De Bruijn numbers (similar to renaming of variables) need not be done at the time of β-reduction but can be done incrementally. This makes the system an attractive basis for designing graph reduction interpreters, for changing of numbers or renaming

of variables implies certain amount of copying. By using λ_f calculus, such copying can be delayed exposing more opportunities for sharing.

Appendix A contains an extension of Barendregt's result. Our result provides some justification for the basic assumption underlying the optimality criterion, *i.e.*, the number of β-contractions performed in reducing in expression is a good measure of the cost of reducing the expression.

In Chapter 6, we also showed that the number of β_{fc} redexes performed by the interpreter in reducing an expression is the same as the number of steps prescribed by Lévy's theory. However, a number of questions arise. First, the interpreter performs not only β_{fc} redexes but other types of redexes. Why do we insist on counting only β_{fc} redexes? The intuitive argument is that it is only these redexes that can create more and possibly unbounded amount of work; the number of other redexes reduced is always bounded. So the question is: Is there a bound on the number of other redexes reduced? Second, how well does the interpreter perform compared to other interpreters? We don't have complete and satisfactory answers to these questions. We will however show some examples and make some general remarks.

Consider the following expression:

$$(\lambda x.(x\, a_1)\,(x\, a_2)\ldots(x\, a_m))\, I$$

A garden variety string reduction interpreter will perform $O(m)$ amount of work in reducing this expression. On the other hand, our interpreter will perform $O(m^2)$ work. The reason is that the first use of I has to deal with one ε and one conditional, the second with two of these and so on. This is the worst case scenario for our interpreter. Thus, the interpreter behaves poorly if there is lots of sharing of a function that has no work that can be shared among various instances of the function. One way to overcome this deficiency of the interpreter is to change the reduction strategy so that ε and conditional introduced by a λ-splitting are propagated as much as possible in order to remove them as soon as possible. The other possibility is to use share contexts only when it pays off; in other cases, simply create a new copy of the shared λ-abstraction.

On the other hand, consider the following expression:

$$M \equiv (\lambda x.(x\, a_1)\,(x\, a_2)\ldots(x\, a_m))$$
$$M\,(\lambda y_1.M\,(\lambda y_2\ldots(\lambda y_k M\,(\lambda q.I^n\,(y_1\, y_2\ldots y_k\, q)))\ldots))$$

Note that because of nested sharing there will be m^k paths to n redexes inside the λq. A rough estimate of the work performed by our interpreter is $O(n + m^{k+1})$. This is actually an over-estimate. On the other hand, a string reduction interpreter or an environment based interpreter will perform *atleast* $O(nm^k)$. Now, if we take $n = m^k$, then, we see the opposite of the worst case for our interpreter. Taking $n = m^k$ is not a ridiculous idea because, in place of I^n, we can use a single redex that can be made to create as much work as desired. Thus, the interpreter performs well compared to other interpreters when there is lots of nested sharing, and there is enough work common to different instances of a shared λ-abstraction.

7.1 Comparison with Related work

Staples [34] proposed an implementation of Lévy's optimal reduction strategy. His interpreter maintains the tree structure of the expression throughout the reduction. The β-reduction simply adds some information to the root of the redex. Neither the reduction strategy nor the translation of the result of reduction to a λ-expression are simple. In a way, it seems that most of the work of reducing a redex is carried out during the translation, which is done not only at the end but repeatedly to find the next redex to reduce.

More recently, John Lamping [20] proposed an interpreter that implements Lévy's optimal reductions. We havn't looked into his interpreter in great detail, but some of the ideas underlying his interpreter are similar to the ideas in our interpreter, especially the use of *fan-out* nodes (conditional nodes in our terminology). His interpreter explicitly keeps track of number of pointers to a node using *fan-in* nodes and uses a number of control nodes to properly match fan-in's and fan-out's. Although his paper doesn't present how to find the leftmost redex, it seems that finding such a redex is not easy; the interpreter has to essentially translate a part of the graph using environment-like structures. [1]

7.2 Some ideas for further research

It would be useful to characterize the class of expressions on which the interpreter performs poorly and the class of expressions on which its performance is good. Also, a comparison of the performance of the interpreter with the performance of other interpreters for the λ-calculus will be useful. The idea of sharing contexts should be examined for implementing other types of

[1] We talked a few times but decided not to talk about our work to avoid conflicts.

languages, *e.g.*, logic languages where it may be useful, for example, in efficiently implementing backtracking. Finally, a better way of measuring the cost of reducing λ-expressions than to count the number of β-steps would be most welcome.

Appendix A

Extension of the result of Barendregt et. al.

Theorem A.1 (Barendregt et. al.) *There is no recursive, optimal reduction strategy for the λ-calculus.*

We will prove a more general result.

Theorem A.2 *For any $k > 0$, there is no recursive reduction strategy that can reduce any λ-expression to normal form (if it exists) in no more than $k \times m$ number of steps where m is the minimum number of steps needed to reduce the expression.*

Before we give the proof, we present some useful definitions and results.

Definition A.3 Let $\mathcal{A} \subset \Lambda$. $\mathcal{A}$ is *closed under equality* if

$$\forall M, N \in \Lambda\, [M \in \mathcal{A} \text{ and } M =_\beta N \Rightarrow N \in \mathcal{A}]$$

Definition A.4 Two sets of non-negative integers $\mathcal{A}$ and $\mathcal{B}$ are *recursively separable* iff there exists a recursive set $\mathcal{C}$ such that $\mathcal{A} \subset \mathcal{C}$ and $\mathcal{B} \cap \mathcal{C} = \emptyset$.

Note that if $\mathcal{A}$ and $\mathcal{B}$ are recursively separable, then they are disjoint.

Notions about sets of integers are translated to terms via a Gödel numbering of terms.

Theorem A.5 (Scott) *Let $\mathcal{A}, \mathcal{B} \subset \Lambda$ be non empty sets closed under equality. Then, $\mathcal{A}$ and $\mathcal{B}$ are not recursively separable.*

Now, the proof. $K \equiv \lambda x.\lambda y.x$, $F \equiv \lambda x.\lambda y.y$.

Proposition A.6 *Suppose P and Q are λ-terms such that their normal forms are of the form $(a\,P_1 \cdots P_n)$. Then,*

1. $M =_\beta K \Rightarrow \mathrm{Min}(M\;P\;Q) = \mathrm{Min}(M) + \mathrm{Min}(P) + 2.$
2. $M =_\beta F \Rightarrow \mathrm{Min}(M\;P\;Q) = \mathrm{Min}(M) + \mathrm{Min}(Q) + 2.$

Proof. Proof is straightforward. A similar proposition appears in [4]. □

Proposition A.7 *For any $k > 0$, let $\mathcal{F}$ be a reduction strategy that can reduce any λ-expression to normal form (if it exists) in no more than $k \times m$ number of steps where m is the minimum number of steps needed to reduce the expression. Let*

$$\begin{aligned} N &\equiv \lambda x.x\;A(a\;\underbrace{x \cdots x}_{n \text{ times}}), \\ P &\equiv \lambda y.y\;B\,(N\,(\lambda u.a\,(u\,B)\,(y\,B))), \text{ and} \\ M &\equiv N\;P \end{aligned}$$

where A, B are in normal form and $n \geq 9k$. Then,

1. *$A\;B =_\beta K$ implies $\mathcal{F}(M)$ is the leftmost redex $N\;P$.*
2. *$A\;B =_\beta F$ implies $\mathcal{F}(M)$ is the redex $N\,(\lambda u.a\,(u\,B)\,(y\,B))$ inside P.*

Proof. By tedious counting. We will use the abbreviation $Q \equiv (\lambda u.a\,(u\,B)\,(y\,B))$. Note that M contains exactly two redexes—the leftmost redex and the redex inside P. Thus, any reduction of M to normal form is one of the following two types:

(i) $\quad M \xrightarrow{lm} M' \longrightarrow \cdots$

(ii) $\quad M \xrightarrow{in\ P} M'' \longrightarrow \cdots$

Proof of (1): In this case, we show that a minimum-length reduction of M to normal form is of the first type, whereas any reduction of the second type has length greater than $k \times \mathrm{Min}(M)$. Thus, if $\mathcal{F}$ is a k-optimal reduction strategy, then it must choose the leftmost redex in M. By assumption and the Church-Rosser theorem $A\,B \twoheadrightarrow K$.

One way to reduce M to normal form in minimum number of steps is as follows:

$$\begin{aligned} \sigma : M &\equiv N\;P \\ &\xrightarrow{lm} P\;A\,(a\,P \cdots P) \\ &\longrightarrow A\;B\,(N\,Q[y := A])\,(a\,P \cdots P) \\ &\twoheadrightarrow \text{normal form of } M \end{aligned}$$

Now, a minimum-length reduction of $N\,Q[y := A]$ to normal form is this:

$$\begin{array}{rcl} N\,Q[y := A] & \twoheadrightarrow & N\,(\lambda u.a\,(u\,B)\,K) \\ & \longrightarrow & C\,A\,(a\,C \cdots C) \text{ where } C \equiv (\lambda u.a\,(u\,B)\,K) \\ & \longrightarrow & a\,(A\,B)\,K\,(a\,C \cdots C) \\ & \twoheadrightarrow & a\,K\,K\,(a\,C \cdots C). \end{array}$$

The length of this reduction is $2 + 2\text{Min}(A\,B)$. We can use Proposition A.6(1) to find the length of σ as the normal forms of $N\,Q[y := A]$ and $(a\,P\,\cdots\,P)$ are of the required form.

$$\begin{array}{rcl} |\sigma| & = & 2 + \text{Min}(A\,B) + \text{Min}(N\,Q[y := A]) + 2 \\ & = & 3\text{Min}(A\,B) + 6 \end{array}$$

Now suppose we reduce the redex inside P first. Then, after that one way to find normal form of M in minimum number of steps is as follows:

$$\begin{array}{rcl} \tau : M & \equiv & N\,P \\ & \stackrel{inP}{\longrightarrow} & N\,(\lambda y.y\,B\,(Q\,A\,(a\,Q\,\cdots\,Q))) \\ & \longrightarrow & P'\,A\,(a\,P'\,\cdots\,P') \text{ where } P' \equiv \lambda y.y\,B\,(Q\,A\,(a\,Q\,\cdots\,Q)) \\ & \longrightarrow & A\,B\,(Q[y := A]\,A\,(a\,Q[y := A]\,\cdots\,Q[y := A]))\,(a\,P'\,\cdots\,P') \\ & \twoheadrightarrow & \text{normal form of } M \end{array}$$

Let $N' \equiv (Q[y := A]\,A\,(a\,Q[y := A]\,\cdots\,Q[y := A]))$. A minimum-length reduction starting with N' is this:

$$\begin{array}{rcl} N' & \longrightarrow & a\,(A\,B)\,(A\,B)\,(a\,\underbrace{Q[y := A]\,\cdots\,Q[y := A]}_{n \text{ times}}) \\ & \twoheadrightarrow & a\,K\,K\,(a\,C\,\cdots\,C). \end{array}$$

The length of this reduction is $1 + (n+2)\text{Min}(A\,B)$. Again, by Proposition A.6(1)

$$\begin{array}{rcl} |\tau| & = & 3 + \text{Min}(A\,B) + \text{Min}(N') + 2 \\ & = & (n+3)\text{Min}(A\,B) + 6. \end{array}$$

Therefore, $n \geq 9k$ implies that $|\tau| > k|\sigma|$.

Proof of (2): In this case, we show that a minimum-length reduction of M to normal form is of the second type, whereas any reduction of the first type has length greater than $k \times \text{Min}(M)$. Thus, $\mathcal{F}$ must choose the redex in P. Note that $A\,B \twoheadrightarrow F$.

Suppose we reduce the leftmost redex in M first. Then, after that one way to find the normal form of M in a minimum number of steps is this:

$$\begin{aligned}
\sigma : M &\equiv N\,P \\
&\xrightarrow{lm} P\,A\,(a\,P\,\cdots\,P) \\
&\longrightarrow A\,B\,(N\,Q[y:=A])(a\,P\,\cdots\,P) \\
&\twoheadrightarrow \text{normal form of } M.
\end{aligned}$$

In this case, we can use Proposition A.6(2) to find the length of σ.

$$\begin{aligned}
|\sigma| &= 2 + \text{Min}(A\,B) + \text{Min}(a\,\underbrace{P\,\cdots\,P}_{n\text{ times}}) + 2 \\
&= \text{Min}(A\,B) + n\text{Min}(P) + 4
\end{aligned}$$

Now, $\text{Min}(P) = 2 + \text{Min}(A\,B)$ as shown by the following reduction:

$$\begin{aligned}
P &\longrightarrow \lambda y.y\,B\,(Q\,A\,(a\,Q\,\cdots\,Q)) \\
&\longrightarrow \lambda y.y\,B\,(a\,(A\,B)\,(y\,B)\,(a\,Q\,\cdots\,Q)) \\
&\twoheadrightarrow \lambda y.y\,B\,(a\,K\,(y\,B)\,(a\,Q\,\cdots\,Q)) \equiv T
\end{aligned}$$

Thus, $|\sigma| = (n+1)\text{Min}(A\,B) + 2n + 4$.

On the other hand, a minimum-length reduction of M to normal form is this:

$$\begin{aligned}
\tau : M &\xrightarrow{inP} N\,(\lambda y.y\,B\,(Q\,A\,(a\,Q\,\cdots\,Q))) \\
&\twoheadrightarrow N\,T \\
&\longrightarrow T\,A\,(a\,T\,\cdots\,T) \\
&\longrightarrow A\,B(a\,K\,(A\,B)\,(a\,Q[y:=A]\,\cdots\,Q[y:=A]))\,(a\,T\,\cdots\,T) \\
&\twoheadrightarrow \text{normal form of } M.
\end{aligned}$$

Again, by Proposition A.6(2)

$$\begin{aligned}
|\tau| &= 1 + (\text{Min}(P) - 1) + 2 + \text{Min}(A\,B) + \text{Min}(a\,T\,\cdots\,T) + 2 \\
&= 2\text{Min}(A\,B) + 6, \quad \text{since } \text{Min}(a\,T\,\cdots\,T) = 0.
\end{aligned}$$

Therefore, $n \geq 9k$ implies that $|\sigma| > k|\tau|$. □

Proof of Theorem A.2: By contradiction. Suppose $\mathcal{F}$ is such a strategy. By Scott's theorem A.5 the disjoint r.e. sets

$$\mathcal{A} = \{M \mid M =_\beta K\}, \qquad \mathcal{B} = \{M \mid M =_\beta F\}$$

are not recursively separable. Now consider a Gödel numbering of *closed* λ-terms. For any closed λ-term M, let $\ulcorner M \urcorner$ denote the Church's numeral corresponding to the Gödel number of M. Then, there exists a closed λ-term E, called *generator*, such that

$$E\ulcorner M \urcorner \twoheadrightarrow M.$$

Furthermore, we may assume that E is in normal form (see Barendregt [4] for details). Since $\mathcal{A}$ and $\mathcal{B}$ are recursively inseparable, so are the disjoint r.e. sets

$$\#\mathcal{A} = \{n \mid E\ulcorner n \urcorner =_\beta K\}, \qquad \#\mathcal{B} = \{n \mid E\ulcorner n \urcorner =_\beta F\}$$

where $\ulcorner n \urcorner$ is the Church's numeral corresponding to the number n.

For each number n, we define M_n to be the term M in Proposition A.7 with $A \equiv E$ and $B \equiv \ulcorner n \urcorner$. Then, the set

$$\mathcal{C} = \{n \mid \mathcal{F}(M_n) \text{ is the leftmost redex in } M_n\}$$

is recursive since $\mathcal{F}$ is a recursive strategy. By Proposition A.7, $\mathcal{C}$ separates $\#\mathcal{A}$ and $\#\mathcal{B}$, which is impossible. □

Bibliography

[1] Arvind, Vinod Kathail, and Keshav Pingali. Sharing of computation in functional language implementations. TR Internal report, Laboratory for Computer Science, M.I.T., Cambridge, MA, July 1984. A preliminary version was published in the Proceedings of the International Workshop on High-Level Computer Architecture, Los Angeles, 1984.

[2] Lennart Augustsson and Thomas Johnsson. Lazy ml user's manual. Technical Report (Preliminary Draft), Programming Methodology Group Report, Department of Computer Science, Chalmers University of Technology and University of Goteborg, S-421 96 Goteborg, Sweden, January 1988.

[3] John 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.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, revised edition, 1984.

[5] H. P. Barendregt, J. Bergstra, J. W. Klop, and H. Volken. Some notes on lambda reduction, in: Degrees, reductions and representability in the lambda calculus. Preprint no. 22, University of Utrecht, Department of Mathematics, 1976.

[6] H. P. Barendregt, J. R. Kennaway, J. R. Klop, and M.R. Sleep. Needed reduction and spine strategies for the lambda calculus. TR CS-R8621, Centrum voor Wiskunde en Infromatica, Ameesterdam, May 1986.

[7] H. P. Barendregt, van Eekelen M. C. J. D., J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *Proceedings of the PARLE Conference, Volume II*, Lecture Notes in Computer Science 259, pages 141–158. Springer-Verlag, June 1987.

[8] G. Berry and J.-J. Lévy. Minimal and optimal computation of recursive programs. *J. ACM*, 26(1):148–175, January 1979.

[9] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, 1941.

[10] G Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[11] P.-L. Curien. *Catgorical Combinators, Sequential Algorithms and Functional Programming.* Research Notes in Theoretical Computer Science. Pitman, London, 1986.

[12] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Proceedings of Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences*, 75:381–392, 1972.

[13] N. G. de Bruijn. Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings. *Proceedings of Koninklijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences*, pages 348–356, 1978.

[14] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science 73, pages 440–455. Springer-Verlag, 1979.

[15] P. Henderson and J. H. Morris. A lazy evaluator. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, pages 95–103. unknown, January 1976.

[16] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[17] Gerard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[18] R.J.M. Hughes. Super-combinators. In *Proceedings of the 1982 Lisp and Functional Languages Conference*, 1982.

[19] J. W. Klop. *Combinatory Reduction Systems.* Centrum, Amsterdam, 1982.

[20] J. Lamping. An algorithm for optimal λ*–calculus* reduction. In *Proceedings of Seventeenth ACM Symposium on Principles of Programming Languages*, pages 16–30, January 1990.

[21] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[22] J.-J. Lévy. An algebraic interpretation of the $\lambda\beta k$-calculus; and an application of a labelled λ-calculus. *Theoretical Computer Science*, 2(1):97–114, 1976.

[23] J.-J. Lévy. *Réductions correctes et optimales dans le lambda calcul.* PhD thesis, Université Paris VII, 1978.

[24] J.-J. Lévy. Optimal reductions in the lambda-calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 160–191. Academic Press, London, 1980.

[25] J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, MA, 1965.

[26] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[27] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[28] J. C. Raoult. On graph rewritings. *Theoretical Computer Science*, 32:1–24, 1984.

[29] M. Ronan Sleep. Issues in implementing lambda languages. Notes for Workshop on Functional Languages, Ustica, Italy, September 1985.

[30] J. Staples. A graph-like lambda calculus for which leftmost-outermost reduction is optimal. In *Graph Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science 73, pages 440–455. Springer-Verlag, 1979.

[31] J. Staples. Computation on graph-like expressions. *Theoretical Computer Science*, 10:171–185, 1980.

[32] J. Staples. Optimal evaluation of graph-like expressions. *Theoretical Computer Science*, 10:297–316, 1980.

[33] J. Staples. Speeding up subtree replacement systems. *Theoretical Computer Science*, 11:39–47, 1980.

[34] J. Staples. Two level expression representation for faster evaluation. In *Graph Grammars and Their Application to Computer Science, Lecture Notes in Computer Science 153*, pages 392–404, Berlin, October 1982. Springer-Verlag.

[35] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* M.I.T. Press, Cambridge, MA, 1977.

[36] D. A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9(1):31–49, January 1979.

[37] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France (Springer-Verlag LNCS 201)*, pages 1–16, September 1985.

[38] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[39] C. P. Wadsworth. *Semantics and pragmatics of the lambda-calculus.* PhD thesis, University of Oxford, 1971.