Computation in director string calculus

Vugranam Chakravarthy Sreedhar

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Computation in director string calculus

Sreedhar, Vugranam Chakravarthy, M.S.
University of Nevada, Las Vegas, 1990
Computation in Director
String Calculus

by

Vugranam Chakravarthy Sreedhar

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science

Department of Computer Science and Electrical Engineering
University of Nevada, Las Vegas
August 1990
The thesis of Vugranam Chakravarthy Sreedhar for the degree of Master of Science in Computer Science is approved.

Chairperson, Dr. Kazem Taghva Ph.D.

Examiner Committee Member, Dr. Thomas Nartker Ph.D.

Examiner Committee Member, Dr. Ajoy Datta Ph.D.

Graduate Faculty Representative, Dr. Ebrahim Salehi Ph.D.

Graduate Dean, Dr. Ronald W. Smith Ph.D.

University of Nevada, Las Vegas
August 1990
Abstract

Computation in Director String Calculus

In this thesis we introduce a modified version of Director String Calculus (MDSC) which preserves the applicative structure of the original lambda terms and captures the strong reduction as opposed to weak reduction of the original Director String Calculus (DSC). Furthermore, our reduction system provides an environment which supports the nonatomic nature of substitution operation and hence can lend itself to parallel and optimal reduction. We shall compare our reduction method with other reduction methods, and discuss some of the advantages and disadvantages of our method.
## Contents

1 Introduction

1.1 Overview ................................................................. 1

1.2 History of Computing ................................................. 2

1.2.1 Role of Logicians .................................................. 2

1.2.2 Role of Computer Scientists ................................. 3

1.3 Functional Approach .................................................. 4

2 Reduction Strategies .................................................. 10

2.1 Previous Proposals ..................................................... 10

2.2 Introduction to Graph Reduction ................................. 12

2.2.1 Representation ....................................................... 12

2.2.2 Evaluation Process ............................................... 13

2.2.3 Graph Sharing ....................................................... 15

2.3 Curry/Turner's Optimization method .......................... 16

2.3.1 Graph Reduction Rules for Combinators .................. 16

2.4 Labeled λ-Calculus and Levy's Optimal Reduction ....... 18
List of Figures

2.1 Representation of internal nodes ................................................... 13
2.2 A redex and its contraction ......................................................... 14
2.3 An example illustrating graph reduction ................................. 14
2.4 An example illustrating graph sharing ....................................... 15
2.5 Combinatory graph rewrite rules .............................................. 17
2.6 An example illustrating director string evaluation ...................... 30

3.1 A director string redex ............................................................. 33
3.2 Tree representation ................................................................. 37
3.3 Left hand side of each binary rule ........................................ 39
3.4 Right hand side of rule I.1 ....................................................... 40
3.5 Right hand side of rule I.2 ....................................................... 41
3.6 Right hand side of rule I.3 ....................................................... 41
3.7 Right hand side of rule I.4 ....................................................... 42
3.8 Right hand side of rule II.1 ...................................................... 42
3.9 Right hand side of rule II.2 ...................................................... 43
3.10 An example ................................................................. 46
3.11 A detailed example ......................................................... 48
3.12 An example illustrating a minor problem in rule I.4 ........ 49
3.13 Result of applying rule I.4 ............................................. 49
3.14 Result of backtracking operation ................................. 51
First of all, I would to thank my advisor Kazem Taghva for inspiring me throughout the course of this thesis. His constant encouragement and stimulating environment made me work more than my capacity. His support, both emotional and professional, were invaluable to my education and research. Next I would like to thank Thomas Nartker for his support and encouragement throughout my education and research at UNLV. Without his support this thesis would not have been possible. I had enlightening discussions with Ajoy Datta throughout my stay in this university. His support and inspiration were of great value. I would like to thank Thomas Nartker, Ajoy Datta, and Ebrahim Salehi for serving on my committee and for their helpful comments.

I would like to thank the Army Research Office for providing me with funds to pursue my advanced education and research.

I would like to thank John Lamping of Xerox and John Field of Cornell University for helping me to understand many aspects of λ-calculus.

I would like to thank my office mate Joe Lombardo for the best time I
had with him sharing the office. We had very interesting and enlightening
discussion on various subject matters. Special thanks to Lynn Lombardo for
helping me out writing this thesis. Her critics and comments were enlight­
ening and beneficial.

I would like to thank persons working at the UNLV library, especially
Thurston Miller, who helped me to find books and other articles in the course
of my research; persons at the circulation desk, rush cataloging and inter-
library loan for tolerating with the "trouble maker".

I would like to thank Ashok Iyer for his encouragement and moral support
during the first semester of my stay at UNLV. I would like to thank Jay
Nietling for helping me out understanding many aspects of system features.
I owe a special thanks to Jagennathan for his moral support. He is both a
mentor and a friend to me.

Most of all, I would like to thank my family; my father for sacrificing
so much for me; both my brothers and their wives for their inspiration and
encouragement; and finally my sister for her support. This thesis is dedicated
to the memory of my mother.

This research was supported in part by U.S. Army Research Office
under grant DAAL03-87-G-0004
Chapter 1

Introduction

1.1 Overview

The basic underlying computational model governing the execution of functional languages is λ-calculus (or one of its derivatives like combinatory logic, equational logic, etc). A number of alternative reduction systems have been proposed in the literature to evaluate λ-terms. [3, 14, 19, 25, 44, 41, 43, 47, 51, 54, 56, 60, 63]. One such reduction system is the Director String Calculus. The problem with this calculus is that the reduction is a weak reduction. In this thesis we shall extend the director string calculus to capture strong reduction.

After a brief introduction to the history of computing in the next section, we shall discuss some aspects of functional programming in section 1.3. In chapter 2 we shall discuss some of the alternative reduction methods proposed in the literature. In chapter 3, which is essentially my thesis, we shall extend the director string calculus to capture strong reduction. We shall also
compare our reduction method with other reduction methods, and discuss some of the advantages and disadvantages of our method.

1.2 History of Computing

1.2.1 Role of Logicians

Until 1931, it was generally accepted that every well defined mathematical problem could be solved “algorithmically”. The great mathematician David Hilbert, a leading proponent of this theory, proposed his famous entscheidungsproblem, or decision problem, in which he hoped that one could solve all mathematical problems in some finite number of steps. Unfortunately, his hopes were shattered. Kurt Gödel, in 1931, published his famous incompleteness theorem which informally states: any consistent formal system that includes the axioms of the theory of numbers is incomplete. In other words, we can always find a number-theoretic statement that can be neither proved nor disproved in the system.

Logicians then began to investigate the characterizations of algorithms or computable functions. They began to ask questions: Of what primitives are all computable functions composed? Alonzo Church and Stephen Kleene, and, independently, Emil Post, proved there are problems that no algorithms can solve. Alan Turing also showed this, independently using his famous “universal machine”. Several systems were offered to formalize the notion of algorithms, among which partial recursive functions, λ-calculus, Post system,
combinatory logic and Turing machines, turned out to be popular. Although they are all equivalent, each one presents a unique approach to computation.\(^1\)

### 1.2.2 Role of Computer Scientists

The Turing machine led directly to the invention of the Von Neumann computer. The invention of computers resulted in the development of programming languages. The λ-calculus possesses several features of programming languages. LISP, for example, as conceived by John McCarthy [57], was heavily influenced by λ-calculus. With the development of programming languages came the question: What is the “meaning” of a program? or What does a program "denote" mathematically? This lead to the development of the theory of formal semantics. Until 1969 nobody could give a consistent meaning to functions represented in a programming language (or even in λ-calculus). Scott, in 1969, constructed λ-calculus models by introducing Scott’s topology on complete lattice [67, 71]. Using the powerful and elegant notation due to Strachey and rigorous theory due to Scott, it became clear how to develop a denotational description for programming languages [71]. Other methods for formal semantics of programming languages were also derived, directly or indirectly, from the theory of recursive functions.

With the development of programming languages came also the questions of structured programming [22], proof of correctness, parallel programming

---

\(^1\)See [20] for an excellent collection on the early development of the theory of computation. See also a recent article by Kleene [50]
[65], limitations of von Neumann computers [2], etc. Procedural languages, like Pascal, have side-effects (for example, assignment statements). Because of side-effects, it is difficult to prove program correctness. Backus in his seminal paper [2] proposed the language FP based on combinatory logic, which overcomes some of the drawbacks of procedural languages. We shall discuss some aspects of functional programming in the next section.

1.3 Functional Approach

The pioneering work of Backus [2], and later Turner [74], lead to the development of functional programming languages. Pure-Lisp, as conceived by McCarthy [57], is probably the first functional programming language to be developed. Later, Landin [14, 54] introduced a language, ISWIM, which contributed many features (both syntactic and semantic) to the development of modern functional languages. APL, by Iverson, had much influence on the development of Backus' FP. Backus' Turing Award lecture [2] was probably the most influential and widely cited literature crediting functional programming. At same time Backus was working on FP, Gordon and his group at Edinburgh were working on a proof generating system called LCF and developed the interactive language ML to serve as the command language for LCF. Later ML was standardized and was called SML [58]. David Turner was working on efficient implementation of functional languages via combinators
which resulted in a series of three languages: SASL, KRC, and Miranda\(^2\) [74, 75, 76]. Turner emphasized higher-order functions, lazy evaluation and other features for functional programming. His work has influenced many of the modern functional programming languages. In September of 1987, a group of well known researchers got together at the conference of Functional Programming and Computer Architecture [46] in Portland, Oregon, to standardize the functional style for programming, resulting in the language called Haskell [40]. Haskell is a large and complex language. It has many innovative features. Only time can tell whether Haskell will be the standard language for the functional style of programming\(^3\).

Before understanding the functional approach, we shall highlight some of the drawbacks of procedural languages:

- The implicit program state in procedural programming is modified by the commands in the source language. This causes side-effects during the program execution. As a result of this, we need a notion of precise sequencing of the commands (for example, for-loops, begin ... end, etc.) to control the states in a deterministic way. The destructive assignment statement is a typical example of a command that causes side-effects.

- Because of side-effects procedural programming is not referentially trans-

\(^2\)Miranda™ is a trademark of Research Software Ltd.

\(^3\)Most of the material in this paragraph is from [38]. For a more detailed discussion on the evolution of functional programming paradigm see [38].
parent, that is, an expression, in general, cannot be replaced by its value without affecting the surrounding expressions.

- The philosophy behind procedural programming is always how to compute, rather than what to compute. The programmer has to know how the program gets evaluated at every step, and therefore program and data abstraction is difficult in procedural programming.

- Writing parallel programs is difficult in procedural programming. The programmer has to identify the tasks that can run concurrently. Then, he has to define the communication and synchronization protocol between these tasks, being aware of hazards (like deadlocks, race-conditions, etc). Finally he has to map the complete task on a particular architecture. Therefore, the programmer has to have the knowledge of the scheduling policies (which, generally, are not accessible to programmers.)

- Program verification is very difficult in procedural programming, both for sequential programming and especially for parallel programming.

- Debugging parallel procedural programs is also very difficult.

Functional languages are languages in which computations are carried out through the evaluation of expressions. Let us now characterize some of the main features of functional languages.
• No side-effects. The assignment-free programming in functional programming languages is analogous to goto-free programming in structured programming languages.

• Higher order functions. Functions are treated as first-class objects. Functions can be returned as a result, passed as arguments, or stored in data structures. Functions have values just like any others. Usually curried notation (also called partial application) is used to represent functions.

• Referential transparency. An expression can be replaced by its value, independent of the surrounding expressions. Because of this property, it is easier to prove program correctness, and also the order of evaluation of expressions does not affect the final result.

• Nonstrict semantics (lazy evaluation). Normal order reduction rule is guaranteed to produce the normal form, if one exists. Normal order evaluation avoids the evaluation of an argument if it is not used in the function body, hence the name lazy evaluation. Unfortunately lazy evaluation is very inefficient. One of the advantages of lazy evaluation is its ability to compute infinite data structure (like infinite list structure).

• Abstraction mechanism. Functional programming languages allow separation of function definition from their use.
• Equations and Pattern matching. Pattern matching and equations go together in functional programming. Because of the lack of side effects we can apply equational reasoning when evaluating functions.

For a detailed discussion on functional programming see [7, 25, 32, 63].

Although functional programming languages are gaining importance, functional language compilers are very inefficient in performance compared to procedural language compilers on conventional Von Neumann machines. Many new and innovative ideas are being proposed to improve the performance of functional language compilers [38]. I/O operations are believed to cause side-effects. Hudak and Sundaresh [42] give a good discussion of how to handle I/O operations in functional programming. Another interesting area of research is Abstract Interpretation [1, 29]; if a function is safe with respect to an argument, then that argument can be evaluated before passing the argument to the function body, without affecting the semantics of the final result. Also, once the safety condition is determined, the arguments to the functions can all be evaluated in parallel. A number of standard and non-standard denotational semantics [9, 10, 11] have been proposed to determine the safety condition. Another problem with functional programming is updating aggregate objects (like arrays) [39, 36, 8]. Arrays are usually defined as a “single-entity”, instead of place-holder that is to be updated incrementally (as in, for example, Pascal). Handling arrays is expensive and tricky in functional programming languages, because destructive updates are impossible.
Other interesting areas that are being investigated are reduction strategies (see chapter 2), graph reduction machines [24, 45, 46, 59, 63], nondeterminism [36], type system [16], parallel functional languages [37], hybrid languages (like functional/logic, functional/object-oriented [12], functional/procedural [62], etc), garbage collection [31], data flow machines, functional operating systems, etc.

We assume that the reader has some basic understanding of $\lambda$-calculus and combinatory logic (see [3, 19, 35] for a detailed discussion and see appendix for a brief introduction).
Chapter 2

Reduction Strategies

2.1 Previous Proposals

The $\lambda$-calculus $\beta$-reduction rule\(^1\) contracts a redex. Other reduction rules like $\alpha$-conversion, $\eta$-conversion rules etc., are mere technicalities [3]. Given a reduction $M \Rightarrow_{R} N$, where $M$ and $N$ are expressions, any reduction method $R$ should be "faithful" to the $\beta$-reduction, in other words, the reduction should satisfy Church-Rosser's property. A number of reduction strategies and implementation methods have been proposed in the literature that are equivalent to the $\lambda$-calculus [3, 14, 19, 25, 44, 41, 43, 47, 51, 54, 56, 60, 63]. Most of the practical implementations of the $\lambda$-calculus are usually described in terms of alternative notion of reduction [4, 5, 14, 15, 25, 43, 27, 48, 53, 60, 63, 66, 68, 69, 74]. The $\beta$-conversion rule, in any standard text book on $\lambda$-calculus [3, 35, 19], is described in a line or so. However, the substitution operation is not as simple. As Field puts it, "It is thus

---

\(^1\)also called as $\beta$-conversion, $\beta$-substitution or $\beta$-contraction rule
*substitution*, not $\beta$-reduction, which complicates both implementations and the analysis of their performance..."[26]. O'Donnell and Strandh also state:

"... There are, however, some disadvantages of using $\lambda$-calculus the way it is normally defined in the literature. The first problem is the $\beta$-reduction itself. The classical way of defining $\beta$-reduction is in terms of substitutions. However, substitution is not a primitive step, in that the amount of work to be done cannot be bounded by some constant ..."[61].

To overcome the problems of substitution, many alternative methods that are equivalent to $\lambda$-calculus have been proposed in the literature. Landin [14, 32] introduced a stack-based machine (SECD machine) to evaluate lambda expressions. Later Wadsworth [77] introduced graph-based evaluation for lambda expression. de Bruijn [21] introduced a variable free notation for $\lambda$-calculus to avoid explicit $\alpha$-conversion. Turner was the first person to implement a practical functional language, SASL, using combinators and graph-reduction [73, 74]. A combinator is nothing but a lambda expression without free variables. Hughes generalized Turner's combinators and introduced *super-combinators* [44, 64]. Hughes abstracted the whole sub-expression which contains free variables, instead of free variables themselves, as was done by Turner. Dijsktra [23], and independently Kennaway and Sleep [48], noticed the problem with Turner's abstraction rules. Turner's abstraction rules loses the original applicative structure of $\lambda$-calculus. Ken-

---

*I sincerely thank Dr. John Field of Cornell University for enlightening me on some of the questions I posed him on the complexity of $\beta$-reduction.*
naway and Sleep [48] introduced director string combinators which preserve the applicative structure of the λ-calculus. Levy [56] introduced parallel reduction of redexes. Levy defined a notion of "family-class of redexes", whose reduction, if done as a unit step, results in "optimal reduction". Levy [55] also introduced labeled λ-calculus to prove that his reduction method is correct. Klop [51] later simplified the labeling scheme. Lamping [53] developed an algorithm to implement Levy’s family class of redexes using "control nodes". Curien [43], using results from category theory, introduced categorical combinators. Field and Teitelbaum [27], using the notion of incrementality, developed an incremental algorithm that takes advantage of functions that compute repeatedly on inputs differing slightly from one another, thus avoiding unnecessary duplication of similar computations. Revesz [66] introduced axioms for the λ-calculus to avoid explicit substitution operation. Thus, we notice that researchers either try to avoid the substitution operation [66, 43, 74], come up with an alternative notion of substitution for practical implementation [63, 5], or come up with an alternative reduction system that is equivalent to the λ-calculus [60, 56, 43].

2.2 Introduction to Graph Reduction

2.2.1 Representation

Lambda-expressions can be evaluated using graph reduction. The basic idea is to rewrite lambda expressions as an abstract syntax tree. The leaves of
the tree are constants, built-in functions or variables. There are two types of internal nodes:

1. **apply node**: A function \( F \) applied to an argument \( x \) is represented as in figure 2.1.

2. **abstraction node**: A lambda abstraction \( \lambda x.M \) is represented as in figure 2.1.

![Figure 2.1: Representation of internal nodes](image)

**2.2.2 Evaluation Process**

The evaluation of a lambda expression consists of two steps:

1. selecting the next redex; and

2. reducing the redex.

We do the above two steps until there are no more redexes to reduce, that is, until the expression reduces to the normal form\(^3\). In general, a redex

\(^3\)We shall always assume, unless otherwise specified, that every expression to be evaluated has a normal form.
in a graph representation looks like figure(2.2). We select the next redex randomly and apply $\beta$-reduction rule\footnote{See the book by Peyton-Jones [63] for a more detailed and practical implementation of graph reduction. As discussed in the book, practical implementations generally yield weak head normal forms.}

![Figure 2.2: A redex and its contraction](image)

**EXAMPLE 2.1** The graph reduction for $(\lambda x \lambda y. + x y)4^2$ is shown in figure(2.3).

![Figure 2.3: An example illustrating graph reduction](image)
2.2.3 Graph Sharing

We can avoid copying of subgraphs (this we do during $\beta$-reduction) by sharing of subgraphs. We can avoid multiple reduction of a redex by overwriting the root of the redex with the result of the reduction of the redex [63, 25]. Levy has shown that there are lambda expressions for which no order of reduction and sharing avoids duplication of redexes. We shall discuss Levy's optimal reduction in section 2.4.

EXAMPLE 2.2 The graph reduction to evaluate the Church's numeral [13]

$$(((\lambda x \lambda y . x (x y)) (\lambda x \lambda y . x (x y))) a) b$$

using sharing is shown in figure(2.4).

Figure 2.4: An example illustrating graph sharing
2.3  Curry/Turner’s Optimization method

The complexity of combinatory expressions produced by the abstraction rules given in the appendix is unacceptably high for practical implementations. Curry introduced the following optimization rules:

\[
S(KE_1)(KE_2) \rightarrow K(E_1E_2)
\]
\[
S(KE_1)I \rightarrow E_1
\]
\[
S(KE_1)E_2 \rightarrow BE_1E_2
\]
\[
SE_1(KE_2) \rightarrow CE_1E_2
\]

Later, Turner [73] further improved the complexity of combinatory expressions by introducing the following combinators:

\[
S'fghx \rightarrow f(gx)(hx)
\]
\[
B'fghx \rightarrow f(gx)
\]
\[
C'fghx \rightarrow f(gx)h
\]

2.3.1  Graph Reduction Rules for Combinators

Combinator expressions contain no \(\lambda\)-terms, thus we have no \(\beta\)-reduction. We represent combinators by their corresponding graph-rewrite or graph-transformation rules. Figure(3.4) shows the transformation rules for S, K, I, B, C and Y combinators.
Most present day compiler implementations for functional languages are based on combinator graph reduction. Combinator graph reduction has the advantage of simplicity of implementation (a simple set of graph rewrite rules for a fixed set of combinators), and also there is no problem of variable name clashing. A number of abstract machines for combinator graph reduction have been proposed in the literature. The principal advantage of an abstract machine is that the control (of combinator reduction sequence) is specified by a sequence of abstract machine instructions. This lends very naturally to (i) hardware implementation of graph traversal (with pipelining), (ii) new memory architecture (specifically suited for graph implementation), and (iii) new machine architecture for highly parallel machines. For a more detailed discussion on combinator graph reduction see [17, 30, 49, 52, 59]
2.4 Labeled λ-Calculus and Levy's Optimal Reduction

We know that the leftmost outermost (normal order) reduction will guarantee to find the normal form whenever one exists (standardization theorem). But the normal order reduction may not be optimal. That is, it will not reach the normal form in a minimum number of steps. Wadsworth's graph reduction shares only subexpressions. Levy [56] argued that one must share not only subexpressions, but pairs of subexpressions and substitutions for free variables in order to get optimal λ-calculus reduction. Levy showed that it is possible to obtain optimal reduction (at least in principle) if we can reduce a set of redexes, called the family class, in parallel. In order to analyze the optimality of reduction, Levy extended the λ-calculus to labeled λ-calculus [55].

2.4.1 Labeled λ-calculus

Levy [55] introduced labeled λ-calculus to trace a reduction sequence. Here we shall present a simplified version of labeled λ-calculus due to Klop [51].

**DEFINITION 2.1** The set of lambda terms ($\Lambda^L$) is defined on a set of labels $L$. Let $L_0 = \{a, b, c, \ldots\}$ be an infinite set of alphabets. Then, $L$ is defined as follows:

\[ a \in L \quad \text{if} \ a \in L_0 \]

\[5\]The present material is from [6]
The set of labeled λ-terms ($\Lambda^L$) is defined as follows:

- $x \in \Lambda^L$
- $(\lambda x. M) \in \Lambda^L$ if $M \in \Lambda^L$
- $(MN) \in \Lambda^L$ if $M, N \in \Lambda^L$
- $M^a \in \Lambda^L$ if $(M^a) \in \Lambda^L$

**DEFINITION 2.2 (labeled β-reduction)** The β-reduction rule, in labeled λ-calculus, is defined as:

$$(\lambda x. M[x])^o N \rightarrow (M[[N]^a])^a$$

That is, each occurrence of $x$ in $M$ is replaced by $N^a$ and the result is labeled by $a$. The label $a$ is called the degree of the redex appearing on the LHS of the β-reduction rule.

Some subterms may have "empty labels". $(M^a)^b$ is simplified to $M^ab$.

**EXAMPLE 2.3** $((\lambda x.(x^d x^c)^e) y^f)^a = (y^j bd y^j c e)^a$

We can transform from unlabeled to labeled λ-calculus and vice versa [28].

**DEFINITION 2.3** Let $M^l$ be a term of $\Lambda^L$. Then $\text{Erase}(M^l)$, the erasure of $M^l$, is the same term in unlabeled λ-calculus, with all the labels erased.
DEFINITION 2.4 Let $M$ be a term of $\Lambda$. Then $M^1 \in \Lambda^L$ is a labeling of $M$ iff $\text{Erase}(M^1) = M$

DEFINITION 2.5 Erasure of a reduction $\text{Erase}(\rho^1)$ is the unlabeled reduction $\rho$, obtained by erasing the labels of all the terms in the reduction and using $\beta$-reduction instead of labeled $\beta$-reduction.

DEFINITION 2.6 The lifted reduction $\text{Lift}(\rho, M^1)$ is a labeled reduction, with initial term $M^1$, in which we contract redexes which are counterparts of redexes contracted in $\rho : M \rightarrow N$, where $M^1$ is the labeled counterpart of $M$.

2.4.2 Levy’s Optimal Reduction

Consider the example shown in figure(2.4). We use sharing to avoid multiple copies of the same lambda-terms. But notice that we just postpone copying of subgraphs (during $\beta$-reduction) by one step. Levy [56] has shown that there are lambda expressions for which no order of reduction and sharing (as discussed in Section 2.2) avoid duplication of lambda terms. How can we avoid duplication of lambda-terms? Levy defines a family of redexes whose reduction, if done as a unit step, is optimal. Here we shall discuss Levy’s optimal reduction strategy for reducing lambda expressions.

DEFINITION 2.7 (Residuals) Residuals of a redex $R$ after a reduction $M \stackrel{S}{\rightarrow} N$, denoted $R/S$, are all those redexes in $N$ which can be “traced back” to $M$.

\textsuperscript{6}Staples [69, 70, 68] has proved that the leftmost-outermost reduction is optimal for combinatory logic.
We can easily determine residuals in labeled λ-calculus. To determine the residuals of \( R \) in \( N \), we write \( R = (\lambda x.P)Q \) as \( R = (\lambda x.P)^aQ \), where \( a \) is the degree of the redex \( R \). Given \( \rho : M \to N \), \( \rho \) can be “lifted” to the labeled case. The residuals of \( R \), denoted \( R/\rho \), are the redexes \( R', R'', \ldots \) in \( N \) with degree \( a \), and are the only ones. Creation of redexes can similarly be expressed neatly using labeled λ-calculus. Given a reduction \( M \to^R N \) and a redex \( S \) in \( N \), we say that \( S \) is created by the \( R \)-contraction if \( S \) is not the residual of any redex in \( M \) and the degree of the created redex \( S \) contains the underlined degree of the creator redex \( R \) as a subword [6].

Next we define the equivalence of reduction sequence. Two reductions, \( \rho \) and \( \sigma \), are equivalent if they have the same initial and final expressions.

More formally:

**Definition 2.8** Let \( \rho \) and \( \sigma \) be the two reductions starting at \( M \). Then \( \rho \equiv \sigma \) iff \( \rho/\sigma = \emptyset^m \) and \( \sigma/\rho = \emptyset^n \) with \( m = |\rho|, n = |\sigma| \), where \( \emptyset^m (\emptyset^n) \) is a reduction consisting \( m (n) \) empty steps, and \( |\rho| (|\sigma|) \) is the number of steps in \( \rho \) (\( \sigma \)).

Next we shall define the family relation (for which the corresponding set is called the family class).

**Definition 2.9** • Redex \( S \) with “history” \( \sigma \) is a copy of the redex \( R \) with “history” \( \rho \), written \( \rho R \leq \sigma S \), iff there is a reduction \( \tau \) such that \( \rho \tau \equiv \sigma \) and \( S \in R/\tau \).
• Two redexes $R$ and $S$ with histories $\rho$ and $\sigma$ are in the same family class, written $\rho R \simeq \sigma S$ iff $\rho R \leq \sigma S$ or $\sigma S \leq \rho R$ or $\rho R \simeq \tau T \simeq \sigma S$ for some $\tau T$.

**LEMMA 2.1** Family relation $\simeq$ is decidable.

We can say that $\rho R \simeq \sigma S$ iff $R$ and $S$ are "created" in the same way along $\rho$ and $\sigma$.

We can express family relation neatly using labeled $\lambda$-calculus.

**DEFINITION 2.10** Let $\rho : M \to N$ be a reduction. Let $l$ be a labeling of $M$ such that each subterm of $M^l$ has a unique atomic label. Let $\text{Lift}(\rho, M^l)$ be the labeled version of $\rho$. Then a redex $R$ in any term of $\rho$ (not necessarily a redex contracted by $\rho$) is a member of the family class $F^l_w$ iff the corresponding redex $R^l$ in $\text{Lift}(\rho, M^l)$ has degree $w$.

Thus we can see that family classes consist of not only residuals of current reduction, but also may consist of "would be" residuals in a different reduction with the same initial and final terms (that is, an equivalent reduction).

Next we define parallel reduction where we reduce some set of redexes at each step. We can represent parallel reduction as $M \overset{F_1}{\to} M_1 \overset{F_2}{\to} M_2 \ldots \overset{F_n}{\to} M_n$ where the $F_i$ are the sets of redexes in $M_i$ contracted in parallel at each step.

**DEFINITION 2.11** A reduction $F_1 F_2 \ldots F_n$ is complete iff, for every $n \geq 1$, $F_n \neq \emptyset$ is a maximum set of redexes such that, for all $R \in F_n$.

---

7See also [28] for a brief discussion on optimality in $\lambda$-calculus.
and $S \in F_n$, $F_1F_2\ldots F_{n-1}R \simeq F_1F_2\ldots F_{n-1}S$.

At each step of a complete reduction, one non-empty family class is contracted.

Next we need the concept of "neededness" of a redex.

**DEFINITION 2.12** A redex $R$ in a term $M$ is called needed if in every reduction of $M$ to normal form some residual of $R$ is contracted.

Since the normal order reduction guarantees to find the normal form if one exist, $R$ is needed in $M$ iff $R$ is contracted in the normal order path of $M$.

**DEFINITION 2.13** A parallel reduction

$$M \xrightarrow{F_1} M_1 \xrightarrow{F_2} M_2 \ldots \xrightarrow{F_n} M_n$$

is a normal order reduction iff there is at least one needed redex in each $F_i$.

**THEOREM 2.1** A complete normal reduction is optimal, that is, reaches normal form, if one exists, in a minimum number of steps.

Levy's parallel reduction is a generalization of Wadsworth's graph reduction. In Wadsworth's graph reduction only residuals of the current reduction are contracted as a unit step. However, Levy's family class consists of not only residuals of the current reduction, but also would be residuals in an alternate reduction. Thus the set of terms contracted in Wadsworth's reduction is a subset of Levy's family classes [26].
Lamping proposed an implementation of Levy's optimal reduction scheme. The complexity of the implementation is so high that the overhead incurred outweighs the advantage of the optimality. See [53] for details.

2.5 Revesz's Axioms for \( \lambda \)-calculus

Revesz [66] came up with a set of axioms for \( \lambda \)-calculus to avoid explicit substitution operation. In this section we shall discuss very briefly his axiomatic approach for evaluation of expressions. The rules for the axiom systems are:

1. \((\lambda x.x)Q \rightarrow Q\)

2. \((\lambda x.P)Q \rightarrow P\) (if \(x\) is not free in \(P\))

3. \((\lambda x.\lambda y.P)Q \rightarrow \lambda z.(\lambda x.P[y := z])Q\) (if \(x \neq y\), \(x\) is free in \(P\), and \(z\) is a new variable)

4. \((\lambda x.(P_1 P_2))Q \rightarrow ((\lambda x.P_1)Q)((\lambda x.P_2)Q)\) (if \(x\) is free in \(P_1, P_2\))

Revesz's axioms correspond to term rewriting systems [60]. Klop\(^9\) has shown that leftmost-outermost reduction is not normalizing for the Revesz's rules. This is because of rule 3, where the beta-redex pushes the incoming argument past the outermost node of the rator. It may so happen that the two incoming arguments can continually "overtake" each other without making progress.

---

\(^8\)I would like to thank Dr. John Lamping for helping me out in the understanding of his algorithm. I had a very productive discussion through email with him.

\(^9\)As noted in [48]
2.6 de Bruijn \( \lambda \)-calculus

de Bruijn [21] introduced a variant of \( \lambda \)-calculus which avoids variable name capturing. The basic idea is to rewrite variables of a lambda term as \( x_0, x_1, \ldots \) where \( i \), called the de Bruijn number, in \( x_i \) is the number of \( \lambda \)s between the variable and the \( \lambda \) which binds it. For instance, \( \lambda x.(\lambda x.xx)(\lambda y.y(\lambda z.x)) \) becomes \( \lambda.(\lambda.(x_0x_0)(\lambda.x_0(\lambda.x_2))\).

The \( \beta \)-reduction can suitably be redefined in de Bruijn \( \lambda \)-calculus. For a concise exposition of de Bruijn \( \lambda \)-calculus see [61].

2.7 Director String Calculus

Turner's translation of \( \lambda \)-calculus to combinatory logic does not preserve the applicative structure of the original lambda terms, and the size of the combinatory term tends to be substantially larger than the corresponding lambda term. Kennaway and Sleep [48] introduced a translation, using director strings, which overcomes the above two problems. In this section we shall briefly discuss the director string calculus.

2.7.1 Motivation

Consider the reduction \((\lambda x.(EF))G \rightarrow (EF)[x := G]^10\). We convey \([x := G]\) to both \(E\) and \(F\), \(E\) alone, \(F\) alone, or neither, depending on the free occurrence of \(x\) in \(E\) and \(F\). This suggests that when we abstract the variable \(x\)

\(^{10}E\) is usually called the rator and \(F\) the rand of the function-body \((EF)\)
from \((EF)\) we can leave *directing symbols* behind and transmit the abstraction process to \(E\) and/or \(F\). To abstract a variable we have the following four rules:

1. \(\lambda x.(E_x F_x) \rightarrow \lambda((\lambda x.E_x)(\lambda x.F_x))\)

2. \(\lambda x.(E_x F) \rightarrow /((\lambda x.E_x)F')\)

3. \(\lambda x.(EF_x) \rightarrow \backslash(E(\lambda x.F_x))\)

4. \(\lambda x.(EF) \rightarrow -(EF)\)

\(\wedge\) sends the argument in both the directions (to both \(E_x\) and \(F_x\)), \(/\) sends the argument only to the left (only to \(E_x\)), \(\\backslash\) sends the argument only to the right (only to \(F_x\)), and \(-\) does not send the argument either direction. For a case of the form \(\lambda x.\lambda y E\), the abstraction takes place innermost out. Such multiple abstractions leave *director strings* appearing at the \(@\)-nodes of the tree. For a variable we have two cases:

1. \(\lambda x.x \rightarrow I\)

2. \(\lambda x.y \rightarrow (K y),\) where \(y \neq x.\)

Notice that the \(K\) combinator introduces a new \(@\)-node, and therefore does not preserve the applicative structure of the original lambda expression. In the next section we shall remove the need for \(I\) and \(K\) combinators, and thus preserving the applicative structure of the original lambda expression.
To evaluate an expression we look at the leading director of the director string. Depending upon the type of director we appropriately send the argument into the function body.

2.7.2 Formal Treatment

In this section we present a formal construction of director string combinatory logic.

DEFINITION 2.14 The director string terms DST are constructed from

1. \{\&, /, \, -\} the binary directors, and the corresponding string is called the binary director string DIR₂.

2. \{!, #\} the unary directors, and the corresponding string is called the unary director string DIR₁.

3. \{Δ\} a hole or a place holder.

4. x, y, z, ... the variables VAR

# is called the discard symbol, and ! the insert symbol.

Let D, D₁, D₂, d, d₁, d₂, @₁ and @₂ denote arbitrary director string, director string over unary operator, director string over binary operator, arbitrary director symbol, unary director symbol, binary director symbol, empty director string over unary operator, and empty director string over binary operators respectively. We define the set MDST of modified director string terms as follows:
\[ (D_1, a) \in DST, \text{ where } a \text{ is either a variable or } \Delta \]

\[ (D_1, E) \in DST, \text{ where } E \in DST \]

\[ (D_2, E_1, E_2) \in DST, \text{ where } E_1, E_2 \in DST \]

Translating \( \lambda \)-calculus to director string calculus

Initially we translate \( \lambda \)-calculus terms to a mixed \( \lambda \)-calculus and director string terms \( DS\lambda \). We replace every variable \( v \) by \( (@_1, v) \) and every application \( (EF) \) by \( (@_2, E, F) \). Then we translate \( DS\lambda \) terms to \( DST \) terms using the following translation rules:

\[ \lambda x.(D_1, x) \rightarrow (!D_1, \Delta) \]

\[ \lambda x.(D_1, E) \rightarrow (#D_1, E) \]

\[ \lambda x.(D_2, E_1, F) \rightarrow (\land D_2, (\lambda x.E_1), (\lambda x.F)) \]

\[ \lambda x.(D_2, E_2, F) \rightarrow (\lor D_2, (\lambda x.E_2), F) \]

\[ \lambda x.(D_2, E, F) \rightarrow (\land D_2, E, (\lambda x.F)) \]

\[ \lambda x.(D_2, E, F) \rightarrow (-D_2, E, F) \]

The evaluation rules for director string terms

The rules for evaluating director string terms are as follows:

\[ (@_2, (\land D_1, E_1, E_2), E_3) \rightarrow (D_1, (@_2, E_1, E_3), (@_2, E_2, E_3)) \]
• $(@_2, (/D, E_1, E_2), E_3) \rightarrow (D, (@_2, E_1, E_3), E_3)$

• $(@_2, (\backslash D, E_1, E_2), E_3) \rightarrow (D, E_1, (@_2, E_2, E_3))$

• $(@_2, (-D, E_1, E_2), E_3) \rightarrow (D, E_1, E_2)$

• $(@_2, (!D, E_1), E_2) \rightarrow (D, (@_2, E_1, E_2))$

• $(@_2, (#D, E_1), E_2) \rightarrow (D, E_1)$

• $(@_2, \Delta, E_1) \rightarrow E_1$

**EXAMPLE 2.4** Consider $(\lambda f.\lambda x.f(x))$. Translating to mixed $\lambda$-calculus:

$$\lambda f.\lambda x.(@_2, (@_1, f), (@_2, (@_1, f), (@_1, x))).$$

Translating to DS term:

$$\lambda f.\lambda x.(@_2, (@_1, f), (@_2, (@_1, f), (@_1, x))) \rightarrow$$

$$\lambda f.(@\backslash, (@_1, f), \lambda x.(@_2, (@_1, f), (@_1, x))) \rightarrow$$

$$\lambda f.(@\backslash, (@_1, f), (@\backslash, (@_1, f), \lambda x.(@_1, x))) \rightarrow$$

$$\lambda f.(@\backslash, (@_1, f), (@\backslash, (@_1, f), (!, \Delta))) \rightarrow$$

$$(\Lambda \backslash, (!, \Delta), (@\backslash, \lambda f.(@_1, f), (!, \Delta))) \rightarrow$$

$$(\Lambda \backslash, (!, \Delta), (@\backslash, (!, \Delta), (!, \Delta)))$$

Given two arguments $a$ and $b$ to the function we can evaluate the DS term as shown in figure 2.6.
Translating director string terms to lambda terms

The rules for translating director string terms back to lambda terms are as follows:

- $(!D, (\lambda x. E)) \rightarrow \lambda x. (D, E)$
• $(\#D, E) \rightarrow \lambda z.(D, E) \ (x \text{ is a new variable})$

• $(\land D, (\lambda z.E), (\lambda y.F)) \rightarrow \lambda z.(D, E[x := z], F[x :=]) \ (z \text{ is a new variable})$

• $(/> D, (\lambda z.E), F) \rightarrow \lambda z.(D, E, F)$

• $(\setminus D, E, (\lambda y.F)) \rightarrow \lambda z.(D, E, F)$

• $(\rightarrow D, E, F) \rightarrow \lambda z.(D, E, F) \ (x \text{ is a new variable})$

• $\Delta \rightarrow \lambda z.x \ (x \text{ is a new variable})$

The director string reduction is correct up to $\beta$-convertability for terms that represent functions. For ground terms (whose normal form contains no $\lambda$s), the director string reduction is strong. Thus director string implementation of $\lambda$-calculus involves weak reduction. In the next chapter (which is essentially my thesis) we shall remove the constraint of weak reduction using a slightly different set of directors.
3.1 Motivation

Kennaway and Sleep [48] introduced the director string calculus to express β-reduction in terms of smaller steps. In the abstract of their paper [48] they state: "... our implementation of the lambda calculus is correct: For lambda terms with a normal form that contains no lambdas (ground terms), the implementation is shown to yield a lambda calculus normal form. For lambda terms whose normal forms represent functions, it is shown that the implementation yields lambda terms that are beta-convertible in zero or more steps to the normal form of the original lambda term. In this sense, our implementation involves weak reduction according Hindley et al [34]."

Consider the lambda expression \((\lambda x.((\lambda y. y y)x))\). The corresponding director string expression \((A, (\slash, (!, A), (!, A)), (!, A))\) can not be reduced further, although it can be reduced in \(\lambda\)-calculus to \(\lambda x.x x\). When we transform
a lambda term to a director string term, we lose some "information" during the abstraction process. Because of this, it is not possible to reduce director string terms to normal form. To see this, consider the figure 3.1; The director $d$ is either: $/$, or $\backslash$, $\Lambda$, or $\ldots$. The director $d$ encodes that the argument should go either left direction or right direction or both direction or neither direction depending on the type of the director. Suppose that $d$ is $/$, it says that the argument should traverse the left direction, but does not explicitly say not to take the right direction. We add this feature to the director string calculus.

\[
\begin{array}{c}
\text{Figure 3.1: A director string redex}
\end{array}
\]

### 3.2 Modified Director String Calculus

Modified Director String Calculus (MDSC) has the same set of unary operators \{!,$\#$,$\Delta$\} and four binary operators \{$/\,$,$\ldash$,$\backslash$,$\ldots$\} together with pair and triple constructors.

**DEFINITION 3.1** Modified director string terms \textit{MDST} are constructed from
1. \{/\,\-\,-\,-\,-\,-\,-\} the binary directors, and the corresponding string is called the binary director strings \(MDIR_2\).

2. \{!,#\} the unary directors, and the corresponding string is called the unary director strings \(MDIR_1\).

3. \{\Delta\} a hole or a placeholder.

4. \(x,y,z\ldots\) the variables \(VAR\)

where \# is called the discard symbol, and \(!\) the insert symbol.

Let \(D, D_1, D_2, d, d_1, d_2, \oplus_1\) and \(\oplus_2\) denote arbitrary director string, director string over unary operator, director string over binary operator, arbitrary director symbol, unary director symbol, binary director symbol, empty director string over unary operator, and empty director string over binary operators respectively. We define the set \(MDST\) of modified director string terms as follows:

- \((D_1,a) \in MDST\), where \(a\) is either a variable or \(\Delta\)

- \((D_1,E) \in MDST\), where \(E \in MDST\)

- \((D_2,E_1,E_2) \in MDST\), where \(E_1,E_2 \in MDST\)

The binary director symbols \{/\,\-\,-\,-\,-\,\,-\,-\} have the natural interpretations send both to left and right, send to left but not right, send to right but not to left, send neither to left nor right. Although we are at liberty to
choose one symbol for each binary operator, we prefer to use two symbols and intuitively think of them as right side and left side of the binary operator. In addition, these two symbol operators make it easier to explain our evaluation strategies. In addition, these two symbols operators make it easier to explain our evaluation strategies.

Translating \( \lambda \)-calculus to modified director string

In order to go back and forth between \( \lambda \)-calculus and MDSC, we utilize a mixed \( \lambda \)-calculus and Director String terms (\( MDSL\)). We basically map each term in \( \lambda \)-calculus to a term in \( MDSL \) by replacing each variable \( x \) by \( (\@_1, x) \) and every application \( (EF) \) by \( (\@_2, E, F) \). This as in [48] embeds the \( \lambda \)-calculus into a system \( MDS\lambda \) generated by the syntactic rules:

- \( E \in MDST \Rightarrow E \in MDSL \)
- \( E \in MDST \Rightarrow \lambda x.E \in MDSL \), where \( x \) is a variable.

Thus \( \lambda f\lambda x.f(fx) \) converts to the mixed term:

\[
\lambda f\lambda x.(\@_2, (\@_1, f), (\@_2, (\@_1, f), (\@_1, x)))
\]

Then we define a translation from \( \lambda \)-terms in \( MDSL \) to \( MDST \), which removes all \( \lambda \)'s and bound variables.

- \( \lambda x.(D_1, x) \rightarrow (!D_1, \Delta) \)
- \( \lambda x.(D_1, \Delta) \rightarrow (#D_1, \Delta) \)
• \( \lambda x. (D_1, y) \rightarrow (\#D_1, y) \), where \( x \neq y \)

• \( \lambda x. (D_2, E_x, F_x) \rightarrow (\backslash D_2, (\lambda x. E_x), (\lambda x. F_x)) \)

• \( \lambda x. (D_2, E_x, F_x) \rightarrow (\backslash \neg D_2, (\lambda x. E_x), (\lambda x. F_x)) \)

• \( \lambda x. (D_2, E_x, F_x) \rightarrow (\neg\neg D_2, (\lambda x. E_x), (\lambda x. F_x)) \)

• \( \lambda x. (D_2, E_x, F_x) \rightarrow (\neg\neg D_2, (\lambda x. E_x), (\lambda x. F_x)) \)

**EXAMPLE 3.1** The term \( \lambda f \lambda x. f(fx) \) will be written as follows:

\[
\begin{align*}
\lambda f \lambda x. (@ 2, (@ 1, f), (@ 2, (@ 1, f), (@ 1, f))) & \rightarrow \\
\lambda f. (\neg \neg, \lambda x. (@ 1, f), \lambda x. (@ 2, (@ 1, f), (@ 1, x))) & \rightarrow \\
\lambda f. (\neg \neg, (\# f), (\neg \neg, (\# f), (!, \Delta))) & \rightarrow \\
(\neg \neg \neg \neg, \lambda f. (\# f), \lambda f. (\neg \neg, (\# f), (!, \Delta))) & \rightarrow \\
(\neg \neg \neg \neg, (!, \# \Delta), (\neg \neg \neg \neg, \lambda f. (\# f), \lambda f. (!, \Delta))) & \rightarrow \\
(\neg \neg \neg \neg, (!, \# \Delta), (\neg \neg \neg \neg, (!, \# f), (\# !, \Delta))) & 
\end{align*}
\]

One may represent the last expression as the tree shown in figure 3.2
Intuitively, the tree explains how the evaluation must proceed, i.e. starting at the root the first argument to this expression will go both ways, the second argument to right but not to left, at the next tree level, for example, the left subtree says insert the first argument and discard the second argument. We will continue to use the tree structure to explain the conversion rules. We also want to point out that the translation does preserve the applicative structure of the original \( \lambda \)-term.

**Proposition 3.1** *The abstraction rules are confluent and terminating. Every \( \lambda \)-term has a unique normal form, and this normal form is a MDS term.*

### 3.3 Conversion Rules

In this section, we will introduce our conversion rules and give detailed examples to illustrate the difference between MDS and the original DS. Before we
formally write our conversion rules, we need three basic operations, namely \textit{insert}, \textit{shift}, and \textit{remove}.

Let $E \in \textit{MDS}$ and $i, j$ be natural numbers, then:

- \textit{shift}(E, i, j) is defined to be a term $F \in \textit{MDS}$ obtained from $E$ by shifting $(i + 1)$th director $j$ places to the right for each director string in $E$.

- \textit{insert}(E, i, j) is also defined to be a term $G \in \textit{MDS}$ obtained from $E$ by inserting $j$ numbers of $-$ between $(i + 1)$ and $(i + 2)$ positions of each binary director string $E$, and inserting $j$ numbers of $#$ between $(i + 1)$ and $(i + 2)$ positions of each unary director string in $E$.

- \textit{remove}(E, i) is defined to be a term $H \in \textit{MDS}$ obtained from $E$ by removing $(i + 1)$st director of every director in $E$.

\textbf{EXAMPLE 3.2} Let $E$ denote $(-\backslash \ -\backslash \ / \ -\dashv \ -(\#\#\!, \Delta), (\!\#\#\!, \Delta))$, then

\begin{align*}
\textit{shift}(E, 1, 1) & = (-\backslash \ -\backslash \ / \ -\dashv \ -(\#!\#, \Delta), (\!\#\#\#, \Delta)) \\
\textit{insert}(E, 1, 1) & = (-\backslash \ -\backslash \ / \ -\dashv \ -(\#\#\#\!, \Delta), (\!\#\#\#\#, \Delta)) \\
\textit{remove}(E, 1) & = (-\backslash \ -\dashv, (\#!, \Delta), (\!\#, \Delta))
\end{align*}

The operation \textit{insert} will be used to avoid variable clashes ($\alpha$-conversion), \textit{shift} will be used to preserve the correspondence between a director and its binding, and \textit{remove} will be used to indicate that a $\beta$-reduction is done and that a particular director will not be needed further.
In order to make our conversion rules easier to understand, we will use tree representation. Basically, we have two types of rules: the binary rules and the unary rules.

I. Binary Rules The left hand side of each binary rule is of the form:

\[(D, (UdV, E_1, E_2), E_3)\]

where \(D, U, V\) are director strings, \(d\) is a single director, and the length of \(D\) is equal to length of the \(U\), i.e., \(|D| = |U|\) (whenever the length of the director string of the left child is bigger than the length of the director string of the parent, we have a redex).

The tree representation is then:

```
     D
    /   \
   UdV   E_3
  /     \
E_1     E_2
```

Figure 3.3: Left hand side of each binary rule

We will use \(p, r\) and \(l\) for parent, right child, and left child respectively, and give computation rules for evaluating parent, left child, and right child director strings.
I.1

\[(D, (U \mid V, E_1, E_2), E_3) \rightarrow \]

\[ (U^p V^p, (U'^l V'^l, \text{shift}(E_1, |U|, |V|), \text{insert}(E_3, |U|, |V|)), \]

\[ (U'^r V'^r, \text{shift}(E_2, |U|, |V|), \text{insert}(E_3, |U|, |V|))) \]

Pictorially, the tree in figure 3.3 converted to the tree in figure 3.4 which says $E_1$ and $E_2$ will take $E_3$ as an argument.

\[ E_1 \quad U^p V^p \quad \text{where } E'_1 = \text{shift}(E_1, |U|, |V|) \]

\[ E_2 \quad \text{shift}(E_2, |U|, |V|) \quad E'_2 \quad \text{shift}(E_2, |U|, |V|) \]

\[ E'_3 = \text{insert}(E_3, |U|, |V|) \]

\[ E_3 \quad \text{insert}(E_3, |U|, |V|) \]

Figure 3.4: Right hand side of rule I.1

I.2

\[(D, (U - V, E_1, E_2), E_3) \rightarrow \]

\[ (U^p V^p, (U'^l V'^l, \text{shift}(E_1, |U|, |V|), \text{insert}(E_3, |U|, |V|)), \text{remove}(E_2, |U|)) \]
where \( E' = \text{shift}(E_1, |U|, |V|) \)

\( E'_2 = \text{remove}(E_2, |U|) \)

\( E'_3 = \text{insert}(E_3, |U|, |V|) \)

Figure 3.5: Right hand side of rule I.2

I.3

\((D, (U - \{V, E_1, E_2\}, E_3) \rightarrow (U^pV^p, \text{remove}(E_1, |U|), (U^rV^r, \text{shift}(E_2, |U|, |V|), \text{insert}(E_3, |U|, |V|)))\)

where \( E'_1 = \text{remove}(E_1, |U|) \)

\( E'_2 = \text{shift}(E_2, |U|, |V|) \)

\( E'_3 = \text{insert}(E_3, |U|, |V|) \)

Figure 3.6: Right hand side of rule I.3

I.4

\((D, (U - \{V, E_1, E_2\}, E_3) \rightarrow (U^pV^p, \text{remove}(E_1, |U|), \text{remove}(E_2, |U|))\)
II. Unary Rules: The left hand side of each unary rule is of the form 

$$(D, (UdV, E_1), E_3) \rightarrow E_3$$

where $D$ is a binary director string, $UdV$ is an unary director string, and $|D| = |U|$.

II.1

$$(D, (U!V, E_1), E_3) \rightarrow E_3$$

Pictorially, figure 3.7 is converted to a tree with one node labelled by $E_3$.

II.2

$$(D, (U\#V, E_1), E_3) \rightarrow (UV, E_1)$$
Pictorially, in figure 3.9, the figure on the left is converted to the figure on the right.

\[
\begin{array}{c}
D \\
\leftarrow U \# V \\
E_1 \\
\end{array} \quad \rightarrow \quad \begin{array}{c}
UV \\
E_1 \\
\end{array}
\]

Figure 3.9: Right hand side of rule II.2

We now explain in detail how to construct director strings \( U^pV^p \), \( U^lV^l \) and \( U^rV^r \). Let \( D = d_1d_2\ldots d_k \), \( U = u_1u_2\ldots u_k \), and \( V = v_1v_2\ldots v_m \). Intuitively, \( d_i \)'s and \( u_i \)'s refer to the distribution of the variables over left and right expressions in an application. Again referring to figure 3.3, observe that when \( d = \backslash / \), a particular variable occurs in both \( E_1 \) and \( E_2 \), and hence \( E_3 \) will be substituted for that variable in both \( E_1 \) and \( E_2 \). Also, we point out that after substitution, the variables of \( E_3 \) now occur in both \( (E_1E_3) \) and \( (E_2E_3) \).

Let \( d_1, d_2, \ldots, d_k \) refer to variables \( x_1, x_2, \ldots, x_k \), then essentially \( d_i \) says how \( x_i \) is distributed over \( E_3 \) and \( (E_1E_2) \). If \( d_i = q- \) (where \( q \) may be \(-\) or \(/\) ), then \( d_i \) says that \( x_i \) does not occur in \( E_3 \). Hence after reduction, \( x_i \) occurs in \( (E_1E_3) \) or \( (E_2E_3) \) only if it occurred in \( E_1 \) or \( E_2 \) before reduction. This gives two cases, depending on whether \( x_i \) occurs in \( E_3 \) or not:

1. if \( d_i = q- \), where \( q \in \{-, /\} \), then \( u_i^p = u_i \) and \( U^r \) and \( U^l \) are defined
as follows depending on $u_i$ (observe that there may be no $U^l$ or $U^r$ in some cases such as I.2, I.3, or I.4):

(a) $u_i = /\$, then $u_i^l = u_i^r = /-$

(b) $u_i = /-$, then $u_i^l = /-$, $u_i^r = ---$

(c) $u_i = -\$, then $u_i^l = ---$, $u_i^r = /-$

(d) $u_i = ---$, then $u_i^l = ---$, $u_i^r = ---$

$V^p = V$

2. if $d_i = q\$, where $q \in \{-, /\}$, then $U^p$ is defined as follows depending on $d$:  

(a) if $d = /\$, then $u_i^p = /\$

(b) if $d = /-$, then $u_i^p = /-$ when $u_i = s-$ and $u_i^p = /\$ when $u_i = s\$, where $s \in \{-, /\}$.

(c) if $d = -\$ then $u_i^p = -\$ when $u_i = -s$ and $u_i^p = /\$ when $u_i = /s$, where $s \in \{-, \}$.

(d) if $d = ---$, then $u_i^p = u_i$

and $U^r$ and $U^l$ are defined as follows depending on $u_i$:

(a) $u_i = /\$, then $u_i^l = u_i^r = /\$

(b) $u_i = /-$, then $u_i^l = /\$, $u_i^r = --$

(c) $u_i = -\$, then $u_i^l = -\$, $u_i^r = /\$
(d) $u_i = --$, then $u_i^l = u_i^r = \backslash$

$V^p = V$

Now again, let $v_1, v_2, \ldots, v_m$ refer to variables $y_1, y_2 \ldots y_m$. We point out that $y_i$'s represent the list of variables which do not occur in $E_3$. Here is a complete set of rules for evaluating $v_i^l$'s and $v_i^r$'s:

(a) if $v_i = / \backslash$, then $v_i^l = v_i^r = /$

(b) if $v_i = / --$, then $v_i^l = / --, v_i^r = --$

(c) if $v_i = - \backslash$, then $v_i^l = --, v_i^r = /$

(d) if $v_i = --$, then $v_i^l = v_i^r = --$

**EXAMPLE 3.3** Consider the $\lambda$-expression $\lambda f. (\lambda x. \lambda y. x(yf)) \lambda p.p$ or equivalently $(/ --, (-/ - --, (\# ! \# , \Delta), (\# ! / \# , \Delta), (! \# \# , \Delta), (\# ! , \Delta)$ in MDSC. Pictorially, we will show the evaluation in detail. For clarity, we will use asterisk to mark the node where the next reduction occurs.
Figure 3.10: An example

This last tree represents the $\lambda$-term $\lambda f \lambda y. (y f)$ which is the normal form of the original $\lambda$-expression.
EXAMPLE 3.4 Evaluation of Church’s numeral.
After few more steps we get

Figure 3.11: A detailed example

**THEOREM 3.1** *MDSC is confluent.*

Remark: Conversion rule I.4 upon application can cause a minor problem
which we address here. Consider the \(\lambda\)-expression and its MDS equivalent expression in figure 3.12.

![Figure 3.12: An example illustrating a minor problem in rule I.4](image)

After applying I.4 at \(*\), we get the tree in figure 3.13.

![Figure 3.13: Result of applying rule I.4](image)

Starting at the root in figure 3.12, the director \(/-\) says that \(x\) occurs in the left subtree, the next node labelled by \(\backslash\) says that \(x\) occurs both in the left and right subtree, the next node labelled by \(-\backslash\) says \(x\) occurs in the right
subtree, and the next node says $x$ occurs neither in the right subtree nor left subtree. Although it causes no problem when the tree is applied to some argument, it is not representing the same expression in $\lambda$-calculus, according to our definition of abstraction rules, after the corresponding $\beta$-reduction. The problem is due to the reduction $(\lambda q.pp)x$. The solution is easy, we start at $*$-node in figure 3.12 and move up the tree towards the root converting each directors of the form $/-$ or $-\backslash$ to $--$. If we encounter a director of the form $\backslash$, then convert $\backslash$ to $-\backslash$ or $/-$ depending on the direction we went up the tree and stop. This has to be done for each director $d$ of the type $-\backslash$ at $*$-node. We call this operation backtracking. Figure 14 ?? shows the result of this process. Now, we apply rule I.4 at $*$-node in figure 3.14. Backtracking and rule I.4 has to be done one after another and cannot be separated.

Figure 3.14: Result of backtracking operation
Now we can apply I.4. The same problem can also be caused by # in rule II.2 and the solution is the same. From now on, we assume the backtracking operation is done when necessary.

3.4 Translating MDS terms to $\lambda$-terms

In this section, we will show how to translate MDS terms back to $\lambda$-terms using the following rules:

1. $\Delta \rightarrow \lambda x.x$, where $x$ is a new variable

2. $(!D, \lambda x.x) \rightarrow \lambda x.(D, x)$

3. $(#D, y) \rightarrow \lambda x.(D, y)$, where $x$ is a new variable

4. $(/\Delta, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E[x := z], F[y := z])$, where $z$ is a new variable

5. $(/\neg D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E[x := z], F)$, where $z$ is a new variable

6. $(\neg \neg D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E, F[y := z])$, where $z$ is a new variable

7. $(\neg \neg D, \lambda x.E, \lambda y.F) \rightarrow \lambda z.(D, E, F)$ where $z$ is a new variable

Rules (1) to (7) convert MDS terms into MDS\(\lambda\) terms. The following two rules convert MDS\(\lambda\) terms into $\lambda$-terms:

1. $(\oplus_1, E) \rightarrow E$
2. \((\otimes_2, E, F) \rightarrow (EF')\)

**EXAMPLE 3.5** Consider the \(\lambda\)-expression

\[
\lambda f.((\lambda x \lambda y.((xf)y))\lambda p.p).
\]

The equivalent MDS expression is

\[
(/-, (-/, --), (--/---, (#!, #, \Delta), (##, \Delta), (#!, \Delta), (#!, \Delta))
\]

We first translate subterms,

\(
(#!, #, \Delta) \rightarrow (#!, #, \lambda x_1.x_1)
\)

\[
\rightarrow \lambda x_2.(#!, \lambda x_1.x_1)
\]

\[
\rightarrow \lambda x_2\lambda x_1.(#, x_1)
\]

\[
\rightarrow \lambda x_2\lambda x_1\lambda x_3.(\otimes_1, x_1)
\]

Similarly,

\[
(#!, #, \Delta) \rightarrow (#!, #, \lambda x_4.x_5.x_6(\otimes_1, x_4),
\]

and

\[
(#!, #, \Delta) \rightarrow \lambda x_7\lambda x_8\lambda x_9(\otimes_1, x_9),
\]

and

\[
(!#, \Delta) \rightarrow \lambda x_{10}\lambda x_{11}(\otimes_1, x_{11}).
\]
For the subterm

\[ (-/ -- --, (#! #, \Delta), (!##, \Delta)) \]

\[ \rightarrow (-/ -- --, \lambda x_2 \lambda x_1, \lambda x_3. (\oplus_1, x_1), \lambda x_4 \lambda x_5 \lambda x_6. (\oplus_1, x_4)) \]

\[ \rightarrow \lambda x_{12}. (-/ -- --, \lambda x_1 \lambda x_3. (\oplus_1, x_1), \lambda x_5 \lambda x_6. (\oplus_1, x_{12})) \]

\[ \rightarrow \lambda x_{12} \lambda x_{13}. (-/ -- --, \lambda x_3. (\oplus_1, x_{13}), \lambda x_6. (\oplus_1, x_{12})) \]

\[ \rightarrow \lambda x_{12} \lambda x_{13} \lambda x_{14}. (\oplus_2, (\oplus_1, x_{13}), (\oplus_1, x_{12})) \]

Similarly, for the subterm

\[ (/ -- / -- \backslash, (-/ -- --, (#! #, \Delta), (!##, \Delta)), (###, \Delta)) \]

\[ \rightarrow (/ -- / -- \backslash, \lambda x_{12} \lambda x_{13} \lambda x_{14}. (\oplus_2, (\oplus_1, x_{13}), (\oplus_1, x_{12})), \lambda x_7 \lambda x_8 \lambda x_9. (\oplus_1, x_9)) \]

\[ \rightarrow \lambda x_{15}. (/ -- / -- \backslash, \lambda x_{13} \lambda x_{14}. (\oplus_2, (\oplus_1, x_{13}), (\oplus_1, x_{15})), \lambda x_9. (\oplus_1, x_9)) \]

\[ \rightarrow \lambda x_{15} \lambda x_{16}. (-/ -- \backslash, \lambda x_{14}. (\oplus_2, (\oplus_1, x_{16}), (\oplus_1, x_{15})), \lambda x_9 (\oplus_1, x_9)) \]

\[ \rightarrow \lambda x_{15} \lambda x_{16} \lambda x_{17}. (\oplus_2, (\oplus_2, (\oplus_1, x_{16}), (\oplus_1, x_{15})), (\oplus_1, x_{17})) \]

And finally,

\[ (/--, (/ -- / -- \backslash, (-/ -- --, (#! #, \Delta), (!##, \Delta)), (###, \Delta)), (#!, \Delta)) \]

\[ \rightarrow (/--, \lambda x_{15} \lambda x_{16} \lambda x_{17}. (\oplus_2, (\oplus_2, (\oplus_1, x_{16}), (\oplus_1, x_{15})), (\oplus_1, x_{17})), \lambda x_{10} \lambda x_{11}. (\oplus_1, x_{11})) \]

\[ \rightarrow \lambda x_{18}. (\lambda x_{16} \lambda x_{17}. (\oplus_2, (\oplus_2, (\oplus_1, x_{16}), (\oplus_1, x_{18})), (\oplus_1, x_{17})), \lambda x_{11} (\oplus_1, x_{11})) \]

Translating to \( \lambda \)-calculus,

\[ \lambda x_{18}. (\lambda x_{16} \lambda x_{17}. ((x_{16} x_{18}) x_{17}) \lambda x_{11}. x_{11}). \]

This is equivalent to the original lambda term up to \( \alpha \)-conversion.
THEOREM 3.2 Every MDS expression $E$ has a unique normal form with respect to these rules. Furthermore, for each $\lambda$-term $E$, if we translate $E$ into MDS term, denoted by $F(E)$, and translate $F(E)$ back to $\lambda$-term, denoted by $B(F(E))$, then $B(F(E)) = E$ (up to $\alpha$-conversion).

Proof:

1. $E = \lambda x_1 \ldots x_n. x$

   \[ B(F(E)) = B((D, \Delta)) \text{ where } D = \#^{i-1}!\#^{n-i} \text{ if } x = x_i \]
   \[ B(F(E)) = B((D, x)) \text{ where } D = \#^n \text{ if no } x = x_i \]
   \[ = E \text{ (up to } \alpha \text{-conversion) } \]

2. $E = \lambda x_1 \ldots x_n. (FG)$

   First consider $F(E)$. First, we rewrite $\lambda x_n. (FG)$ giving $(d_n, \lambda x_n.F, \lambda x_n.G)$.

   $d_n$ is $\backslash, /-, -, \backslash$, or $-\backslash$, depending upon the free occurrence of $x$ in $F$ and/or $G$. Next, we reduce

   \[ \lambda x_{n-1}(d_n, \lambda x_n.F, \lambda x_n.G) \]

   giving

   \[ (d_{n-1}d_n, \lambda x_{n-1}x_n.F, \lambda x_{n-1}x_n.G) \].

   Continuing this process for all lambdas in the outer most parentheses, we get

   \[ F(E) = F((D, \lambda x_1 \ldots x_n.F, \lambda x_1 \ldots x_n.G)) \]
   \[ = (D, F(\lambda x_1 \ldots x_n.F), F(\lambda x_1 \ldots x_n.G)). \]
Hence, $B(F(E)) = B((D, F(\lambda x_1 \ldots x_n.F), F(\lambda x_1 \ldots x_n.G))$. By confluence of the $B$-rules, we can evaluate the right hand side by first evaluating each subexpression giving

$$B(F(E)) = B((D, F(\lambda x_1 \ldots x_n.F), F(\lambda x_1 \ldots x_n.G))$$
$$= B((D, B(F(\lambda x_1 \ldots x_n.F)), B(F(\lambda x_1 \ldots x_n.G)))$$
$$= B((D, \lambda x_1 \ldots x_n.F, \lambda x_1 \ldots x_n.G)).$$

We can now apply $B$-rules to eliminate $D$, giving

$$B((D, \lambda x_1 \ldots x_n.F, \lambda x_1 \ldots x_n.G)) = \lambda x_1 \ldots x_n.(FG) = E.$$

\[\square\]

### 3.5 Comparison to Related Work

In this section we compare MDSC to other related reduction system discussed in chapter 2.

#### 3.5.1 Curry/Turner's Combinators

The Curry/Turner abstraction rules has the following undesirable properties:

- The translation does not preserve the applicative structure of the original lambda term
- The size of the combinator term tends to be substantially larger than the size of the original term.
• The translation disables some of the redexes present in the lambda term, preventing the possibility of parallel reductions.

The modified director string calculus overcomes the above undesirable properties.

3.5.2 Labeled λ-calculus and Levy’s Optimal Reduction

Let \( M \rightarrow^S N \) be a reduction by contracting the redex \( S \) in λ-calculus.

Let \( R \in M \). Let \( \rho_d : M_d \rightarrow^{S_d} N_d \) be the corresponding reduction in MDSC.

Let \( M \equiv M_d \), and \( N \equiv N_d \). Let \( R_d \in M_d \) be the corresponding redex as \( R \in M \) A \( \beta \)-reduction is equivalent to a sequence of MDS reduction. To see this, let us rewrite \( S_d \) as \( S_d^1S_d^2 \ldots S_d^k \). \( S_d^i \)'s are the redexes generated during the reduction \( \rho_d : M_d \rightarrow N_d \). \( S \) is same as \( S_d^1 \) and \( S_d^k \) when contracted gives \( N_d \). The residuals of \( R_d \) in \( N_d \) is same as the residuals of \( R \) in \( N \). All the residuals of \( R_d \) in \( N_d \) will have the same director strings, except for some extra —— directors introduced during \textit{insert}() operation. We can interpret directors as \textit{labels}. Our reduction systems has the potential for implementing Levy’s optimal reduction. Further investigation needs to done along this line.

3.5.3 Revesz’s Axioms

Our rules for evaluating MDS expressions are related to Revesz’s axioms. Klop has shown that leftmost outermost is not normalizing for Revesz’s rules. This is not true in MDSC.
3.5.4 de Bruijn $\lambda$-calculus

In MDSC the position of a director in a director string plays a crucial role in evaluation of expressions. The position of a director is related to the de Bruijn number. To see the relation, let $i$ be the position of a director in the director string at a node, say, $\omega$-node. Let $N$ be the length of the director string. We encounter $n$ $\lambda$s traversing from the root node to the $\omega$-node. If director string at $\omega$-node are unary, then $(N - i)$ is the de Bruijn number. We illustrate this using the following example.

**EXAMPLE 3.6** Consider the lambda expression $\lambda x. (\lambda x.x) (\lambda y. (\lambda z. x))$.

The equivalent expression using de Bruijn notation is

$$\lambda. (\lambda(x_0x_0)(\lambda.x_0(\lambda.x_2)))$$

and the corresponding MDSC expression is

$$(-\backslash, (- - /\backslash, (#!\Delta), (#!\Delta)), (-\backslash -, (#!\Delta), (!# #\Delta)))$$

The de Bruijn number for the third instance of the variable $x$ is 2. The length corresponding unary director string, highlighted in the above expression, is 3. The position of $!$ in the unary string is 1. Hence, $3 - 1 = 2$, which is equal to the de Bruijn number 2.

The relative position of a variable with respect to its $\lambda$ binding is more important than the variable name. Both, de Bruijn $\lambda$-calculus and MDSC, effectively use this property and avoid explicit $\alpha$-conversion.
Chapter 4

Conclusion and Future Work

We have modified the director string calculus to obtain strong reduction as opposed to weak reaction given in [48]. In addition, this modified calculus can be considered as a different implementation of $\lambda$-calculus. Particularly, the substitution operation supports the idea given by Revesz in [66] and O'Donnell and Strandh in [61]. We can make the following observations regarding our calculus:

- The modified director string preserves the *applicative structure* of the original lambda terms, and does not increase the number of subterms in the translation.

- The translation from $\lambda$-calculus to MDSC do not disable any redex present in the original lambda expression. Thus, MDSC is well suited for parallel reduction as opposed to Curry/Turner combinatory logic and Kennaway and Sleep director string calculus.
• Our reduction rules for contracting a redex do not require an explicit \(\alpha\)-conversion rule. This is an important property because we have removed the problem of \textit{capturing of global variables} after a reduction step.

• In our binary rules, the left and right child can be evaluated in parallel.

The possible future work that can be done are:

• Practical implementations of Kennaway and Sleep' calculus are already in use [72]. We can implement our calculus similarly.

• We can incorporate sharing (similar to Wadsworth's graph reduction) in our calculus.

• All the residuals of a redex have a substring of directors which is the same. This observation tells us that there is a possibility of implementing Levy's optimal reduction in our calculus. Further investigation can be done along this line.

• de Bruijn notation is used in the implementation of Categorical Abstract Machine [18]. MDSC is related to de Bruijn \(\lambda\)-calculus. It is possible that we can extend MDSC to the framework of categorical combinators.
Appendix A

Introduction to $\lambda$-Calculus and Combinatory Logic

A brief introduction to $\lambda$-calculus and combinatory logic is presented (see [3, 19, 35, 51] for a more complete treatment).

A.1 $\lambda$-Calculus

DEFINITION A.1 $\lambda$-terms are strings over:

1. $x_1, x_2 \ldots$ variables

2. $\lambda$ lambda abstractor

3. ( ) delimiters

The set of $\lambda$-terms $\Lambda$ is defined as follows:

- $x \in \Lambda$, where $x$ is a variable

- $M \in \Lambda$ implies $(\lambda x.M) \in \Lambda$
• $M, N \in \Lambda$ implies $(MN) \in \Lambda$

We will omit unnecessary brackets and assume left association for terms and inner-most out for lambda abstractions.

**DEFINITION A.2 (Free and bound Variable [3])** A variable $x$ occurs free in a term $M$ if $x$ is not in the scope of a $\lambda x$; $x$ occurs bound otherwise.

**DEFINITION A.3 ($\beta$-reduction)** $(\lambda x. M)N \rightarrow M[x := N]$. Free variables in $N$ should not be "captured" after the reduction.

We can avoid the problem of free variable $N$ being captured after a reduction by applying $\alpha$-conversion rule in $M$ before a $\beta$-reduction.

**DEFINITION A.4 ($\alpha$-conversion)** $\lambda x. M \rightarrow \lambda y(M[x := y])$, where $y$ is a new variable.

We represent functions in curried notation, where a function gets applied to only one argument, and the result of the application is another function. We consider constants and variables as function with zero argument.

**DEFINITION A.5** A redex (reducible expression) is a term of the form $(\lambda x. M)N$.

**DEFINITION A.6** A term $M$ is in normal form iff there is no subterm of $M$ which is a redex.
DEFINITION A.7 A term $M$ has a normal form iff for some $N$ in normal form $M$ can be reduced to $N$, that is, $\lambda \vdash M \rightarrow N$.

DEFINITION A.8 (Strong reduction) A term $M$ is strongly normalizing iff every reduction of $M$ should (eventually) lead to a normal form; otherwise it is weakly normalizing.

THEOREM A.1 (Church-Rosser) If $\lambda \vdash M \rightarrow N$ and $\lambda \vdash M \rightarrow N'$ then there exists $P$ such that $\lambda \vdash N \rightarrow P$ and $\lambda \vdash N' \rightarrow P$

We also say that $\lambda$-calculus is confluent with respect to the $\beta$-reduction rule (and $\alpha$-reduction rule), or the $\beta$-reduction rule satisfies the diamond-property.

COROLLARY A.1 If $M$ has a normal form then $M$ has a unique normal form.

EXAMPLE A.1 $(\lambda x.x y)(\lambda x.x y)$ does not have a normal form

DEFINITION A.9 1. A leftmost redex is left of all redexes within an expression.

2. An outermost redex is a redex not contained within any other redex.

3. An innermost redex is a redex contains no other redexes.

DEFINITION A.10 (Reduction order) 1. Applicative order Reduce the leftmost innermost redex first. Also called as eager evaluation or call-by-value evaluation.
2. Normal order or Standard order Reduce the leftmost outermost redex first. Also called lazy evaluation or call-by-need evaluation.

**DEFINITION A.11 (Weak Head Normal Form)** An expression $E$ is said to be in WHNF if:

- $E$ is a constant
- $E$ is of the form $\lambda x.M$

We can extend the CR-property to WHNF. If $\lambda M \vdash_w N$ and $\lambda M \vdash_w N'$ then there exist a $P$ such that $\lambda P \vdash_w P$ and $\lambda N \vdash_w P$. Also if $M$ has a WHNF, then $M$ has a unique WHNF.

We have another normal form, called Head Normal Form (HNF), defined as follows:

**DEFINITION A.12** An expression $E$ is said to be in HNF if:

- $E$ is a constant,
- $E$ is of the form $\lambda x_1 \ldots \lambda x_n.M$ where $M$ is not a redex.

Anything in HNF is also in WHNF, but not vice versa.

**THEOREM A.2 (Standardization theorem)** If an expression has a normal form, then leftmost outermost reduction process is guaranteed to attain the normal form.
A.2 Combinatory Logic

DEFINITION A.13 Combinatory logic terms are words over the alphabet

- \( K, S \) constants called combinators
- \( x, y, \ldots \) variables
- ( ) delimiters

The set of combinatory logic term \( C \) is defined as follows:

1. \( x \in C \)
2. \( K, S \in C \)
3. \( P, Q \in C \) implies \( (PQ) \in C \)

DEFINITION A.14 (Basic combinators) \( S, K \) combinators are defined as:

- \( Kxy = x \ \forall x, y \)
- \( Sfgx = f(xg) \ \forall f, g, x \)

We also have the following derived combinators:

\[
\begin{align*}
B_{fgx} &= f(gx) \\
C_{fgx} &= fxg \\
I_{x} &= x \\
W_{fx} &= fxx
\end{align*}
\]
We can show

\[ I = SKK \]

\[ Ix = x \]

\[ SKKx = Kx(Kx) = x \]

Similarly we can show

\[ B = S(KS)K \]

\[ C = S(BBS)(KK) \]

\[ W = SS(KI) \]

**THEOREM A.3 (Fixed point theorem)** *For every term* \( F \), there is a term \( X \) such that

\[ FX = X \]

**DEFINITION A.15 (Fixed point combinator)** A fixed point combinator is a term \( Y \) such that for all \( F \)

\[ YF = Y(YF) \]

\( Y \) can be shown to be \( WS(BWB) \).

Lambda expressions can be translated, by *abstracting* the variables, to combinators. The translation should preserve the following property

\[ ([x]E)x = E \]
where \([x]E\) "abstracts \(x\) from the expression \(E\)". We can see that abstraction is an exact inverse of application.

**DEFINITION A.16 (Abstraction)** For each term \(M\) and variable \(x\), abstraction of the variable \(x\) from the term \(M\) is defined as follows:

1. \([x]x = I\)

2. \([x]M = KM\) if \(x \notin M\)

3. \([x]Ux = U\) if \(x \notin U\)

4. \([x]UV = S([x]U)([x]V)\) if none of the above applies.

**DEFINITION A.17 (Weak reduction [33, 34])** A weak redex in a term \(M\) is any occurrence of a term of the form \(KXY\) and/or \(SXYZ\). A weak reduction \(M \rightarrow^w N\) is the process of contracting weak redexes.

**EXAMPLE A.2** \(SK\) is in weak normal form, but the corresponding lambda term \((\lambda x \lambda y \lambda z.xz(yz))(\lambda x \lambda y.x)\) has a redex \([3]\).
Bibliography


[67] **Scott, D. S.** Lectures on a mathematical theory of computation.


[72] **Stoye, W.** Director strings on SKIM. In *Proceedings of the Functional Programming Workshop* (Göteborg, 1985).


[75] Turner, D. A. Recursion equation as a programming language. In
